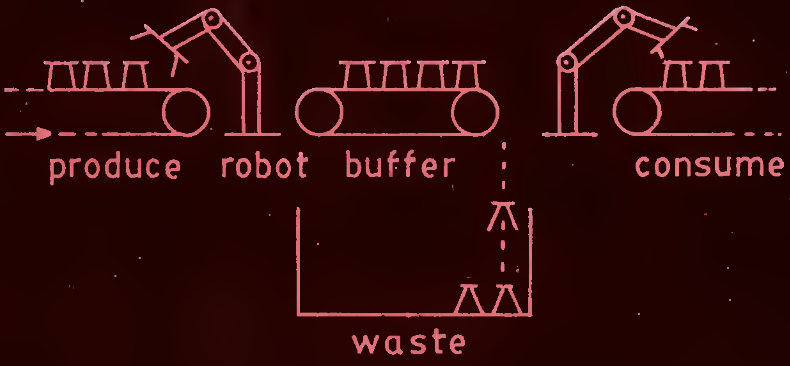
Parallel and Real-Time Distributed Computing

Edited by
R K SHYAMASUNDAR



INDIAN ACADEMY OF SCIENCES
Bangalore 560 080

Parallel and Real-Time Distributed Computing

Parallel and Real-Time Distributed Computing

Edited by
R K SHYAMASUNDAR
Tata Institute of Fundamental Research, Bombay

1992
INDIAN ACADEMY OF SCIENCES
Bangalore 560 080

The cover shows a schematic diagram of an automated machine shop.

Reprinted from *Sādhanā* – Academy Proceedings in Engineering Sciences
Volume 17, pp. 1–220, 1992

Edited by R K Shyamasundar, phototypeset at Thomson Press (India) Ltd., New Delhi and printed for the Indian Academy of Sciences at Macmillan India Press, Madras 600 041

CONTENTS

Parallel and Real-Time Distributed Computing

Foreword

Advances in hardware technologies have led to extensive use of sophisticated processors to build multiprocessors and distributed processors for a variety of real-time systems such as home appliances, flexible manufacturing systems, process control systems, flight control systems and tactical control systems. This in turn has led to challenges in programming distributed, parallel and real-time systems. While distributed programming has been based on the paradigm of achieving a common goal from partial knowledge (information), the principle objective of parallel processing has been to achieve better efficiency and throughput. Even though concurrency has been investigated for many years, it is only recently that a firm formal foundation of real-time programming has been established. A firm mathematical foundation has made it possible to separate the issues of functional and timing correctness of real-time programs. Such a separation in turn has led to a serious investigation of design, methodology and standardization.

The primary purpose of this book is to present:

1. issues and principles of specification and verification of real-time distributed programs;
2. issues of true concurrency;
3. study of epistemology (or the study of knowledge) and distributed systems; and
4. issues of building and programming scalable concurrent computers.

The book is essentially in two parts: the first part is devoted to issues of real-time programming and the latter part covers various aspects of true concurrency, epistemology and scalable concurrent computing.

The first part has four chapters. The first chapter *Modelling real-time systems: Issues and challenges* by R K Shyamasundar and S Ramesh surveys the issues and challenges that lie in the specification, development, and verification of real-time systems. This section mainly emphasizes the issues of real-time distributed concurrency.

Chapter 2 by J J M Hooman and W P de Roever provides an introduction to compositional methods for concurrency and their application to real-time. This section is devoted to a discussion of formal methods to specify and verify concurrent programs with synchronous message passing. The emphasis is mainly on compositional methods, i.e., methods in which the specification of a compound program can be inferred from specifications of its constituents without reference to the internal structure of those parts. Compositionality enables verification during the process of top-down design (the derivation of correct programs) instead of the more familiar a posteriori verification based on an already completed program code. The chapter also highlights the main principles behind compositionality by discussing transitions from non-compositional methods to compositional methods for concurrent programs.

Priority specification plays a vital role in the development of predictable systems. This is discussed in the next chapter by R K Shyamasundar and L Y Liu. The notion of priority is based on the intuition that a low priority action can proceed only if the high priority action cannot proceed due to lack of a handshaking partner at that point of execution. The authors discuss the issues of compositional specification of prioritized real-time distributed programming languages and describe an approach wherein one can preserve compositionality without placing unnecessary restrictions between prioritized events (local or global) and unprioritized events (local or global).

Chapter 4 by G Berry is concerned with ESTEREL, which is one of the highly developed languages of the class of synchronous concurrent languages dedicated to reactive systems. ESTEREL can be distinguished from the languages discussed in the previous chapters by the fact that it is based on the notion of a perfect real-time machine (i.e., synchrony hypothesis, wherein control and communication are assumed to be taking no time). Berry discusses a hardware implementation of a subset of ESTEREL. In the translation described, each program generates a specific circuit that responds to any input in one clock cycle. It is shown that whenever the source program satisfies some statically verifiable dynamic properties, the circuit is semantically equivalent to the source program. It is of interest to note that the translation has been effectively implemented on the programmable active memory Perle0 developed by J Vuillemin and his group at Digital Equipment.

The second part begins with a survey of models and logics for true concurrency by Kamal Lodaya, Madhavan Mukund, R Ramanujam and P S Thiagarajan. In this chapter, the authors first survey formal models of distributed systems in which concurrency is specified explicitly, in contrast to more traditional approaches where concurrency is represented implicitly as a nondeterministic choice between all possible sequentializations of concurrent actions. In the second half of the presentation, the authors develop a family of logics to specify and reason about the behavioural properties of the models described in the first part. The logics defined are extensions of temporal logic with new modalities to directly describe concurrency.

Chapter 6 by Rohit Parikh and Paul Krasucki is devoted to epistemology or the study of knowledge. In this chapter, the authors define various notions of the study of knowledge and distributed systems and introduce the notion of levels of knowledge. The authors discuss how levels of knowledge can be realized in distributed systems and arrive at protocols that precisely realize levels of knowledge of some formulae.

In the last chapter, Nalini Venkatasubramanian, Shakuntala Miriyala and Gul Agha focus on the challenges in building and programming scalable concurrent computers. The chapter brings out the inadequacy of current models of computing for programming massively parallel computers and discuss three universal models of concurrent computing, developed respectively by programming-, architecture- and algorithmic-perspectives. It is shown that these models provide a powerful representation for parallel computing and are shown to be quite close. The authors also argue that by using a flexible universal programming model, an environment supporting heterogeneous programming languages can be developed.

Thanks go to Professors R Narasimha and N Viswanadham for their enthusiasm in bringing out this book which is essentially the result of their ideas and encouragement. I also thank all the authors for the articles and their cooperation in attending to various modifications in a timely fashion and the reviewers for giving the feedback within a short time. Last but not least, I thank the editorial staff of the Indian Academy of Sciences for their help in bringing out the book.

June 1992

R K Shyamasundar
Guest Editor

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 equivalcnce; 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 behaviour 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.

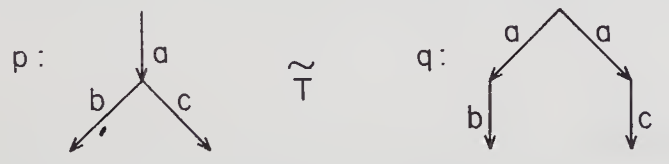


Figure 1. Trace equivalence.

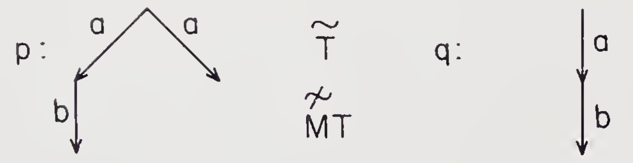


Figure 2. Complete trace equivalence.

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, $F(p) \neq F(q)$.

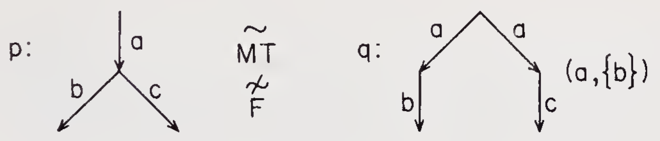


Figure 3. Failure set equivalence.

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.

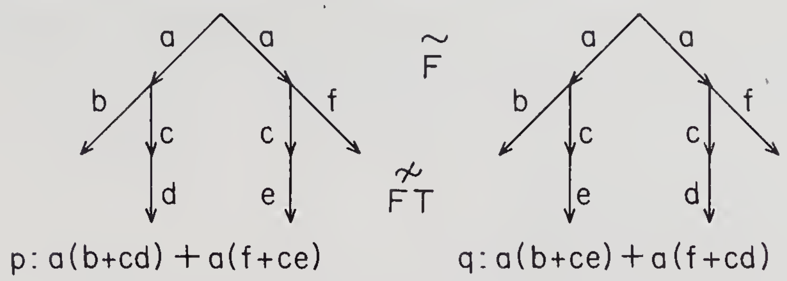


Figure 4. Failure trace equivalence.

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,

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

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.

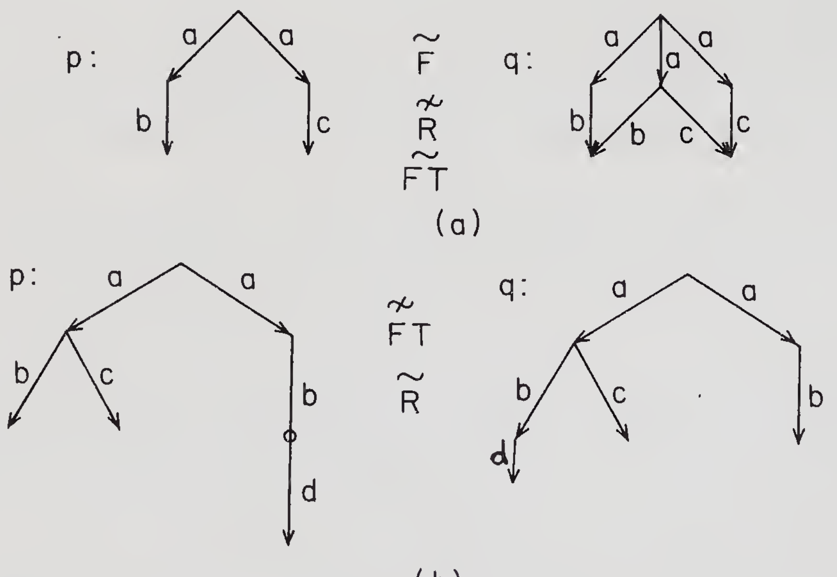


Figure 5. (**a**) Ready set equivalence. (**b**) Ready set and failure trace equivalence.

[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, $\langle V_2,E_2\rangle$) 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.

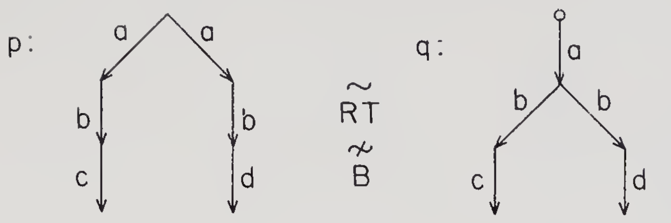


Figure 6. Bisimulation and ready trace equivalence.

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 $Bis(p) = \{\langle a, Bis(q)\rangle \mid p \to^{a} q\}$,

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 $p \not\sim^{\mathrm{MT}} q$. 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.

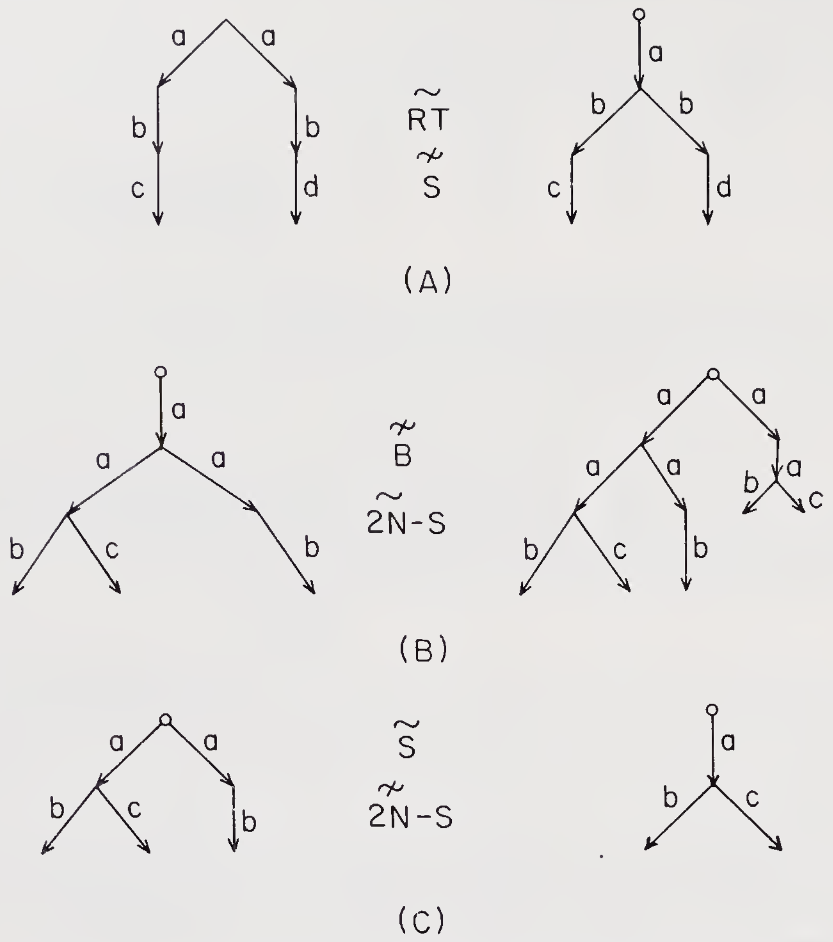


Figure 7. (**a**) Ready trace and simulation equivalence. (**b**) Bisimulation and 2-nested simulation. (**c**) Ready simulation and 2-nested simulation equivalence.

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?

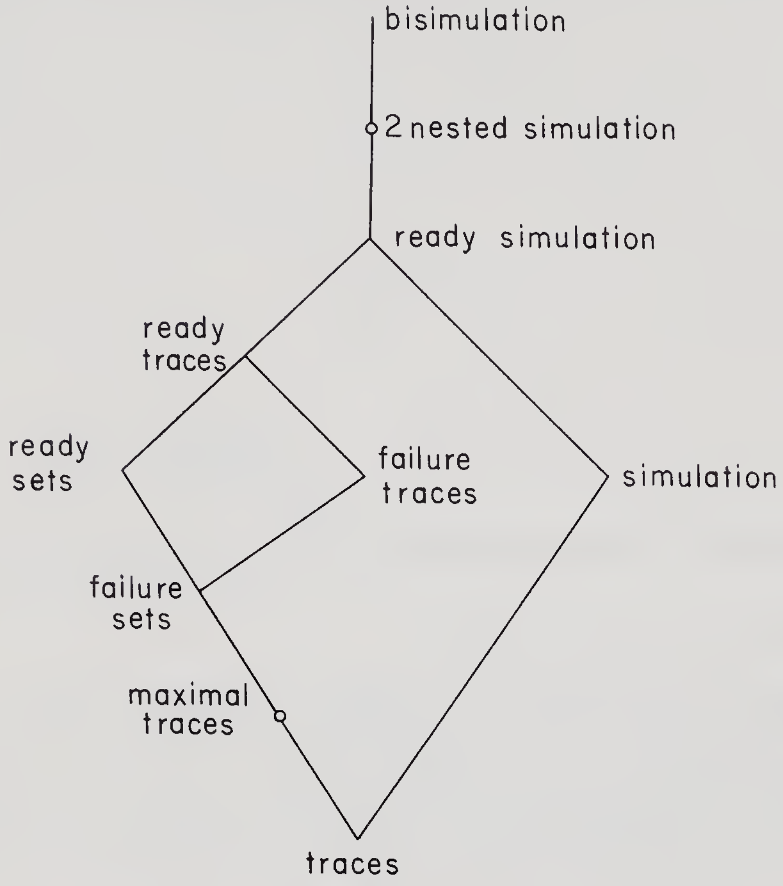


Figure 8. Relationship among various equivalences.

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*

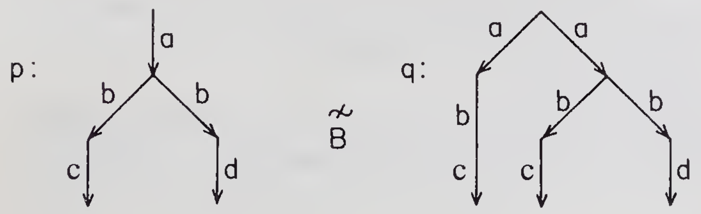


Figure 9. Bisimulation and observational equivalence.

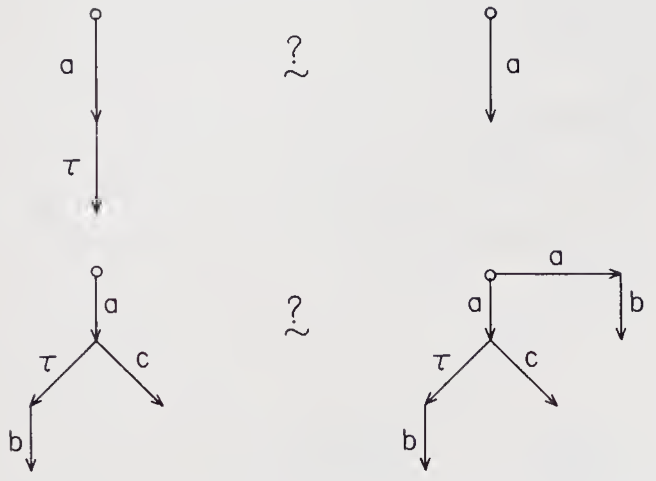


Figure 10. Effect of τ actions on equivalences.

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

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

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

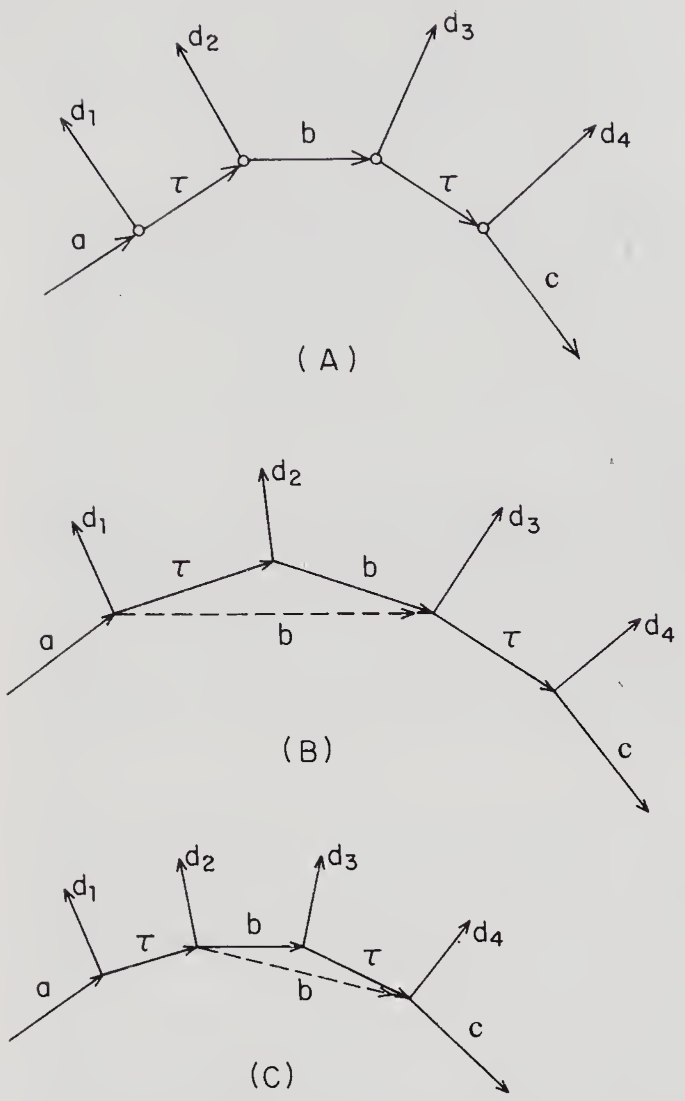


Figure 11. Observation equivalence in branching time.

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

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

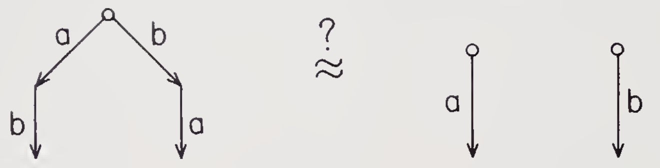


Figure 12. Interleaving vs. true concurrency.

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

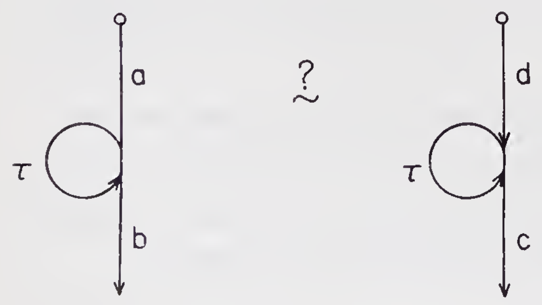


Figure 13. Effect of divergence on equivalence.

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, $\mathscr{U}$, 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 $\mathscr{U}$. 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 $\mathscr{U}$ (symbol $\rightarrow$ reads *precedes*). Two successive state changes $[false \rightarrow true]$ for a given entity in $\mathscr{U}$ correspond to two infinitesimally close points in $\mathscr{U}$'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, $\mathscr{U}$, it is necessary to assume that every change of $t(k)$ is communicated instantaneously to all entities in $\mathscr{U}$. 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 $\mathscr{U}$'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 $\mathscr{U}$'s spacetime topological properties:

- Approximate some unique time dimension throughout $\mathscr{U}$ 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 $\mathscr{U}$ 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* $\rightarrow$ *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.

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

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

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

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

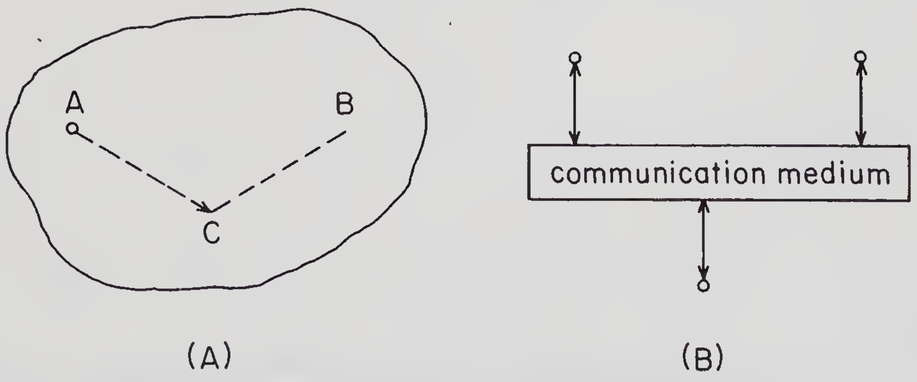


Figure 14. Effect of topology and communication medium.

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

Sādhanā, Vol. 17, Part 1, March 1992, pp. 29–73. © Printed in India.

An introduction to compositional methods for concurrency and their application to real-time

J J M HOOMAN[1] and W P de ROEVER[2]

[1]Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands
[2]Institute für Informatik and Praktische Mathematik II, Christian-Albrechts-Universität zu Kiel, Preusserstrasse 1–9, 2300 Kiel 1, Germany

Abstract. Formal methods to specify and verify concurrent programs with synchronous message passing are discussed. We stress the development towards compositional methods, i.e. methods in which the specification of a compound program can be inferred from specifications of its constituents without reference to the internal structure of those parts. Compositionality enables verification during the process of (top-down) design – the derivation of correct programs – instead of the more familiar a-posteriori verification based on already completed program codes. We sketch the transition from non-compositional towards compositional methods for concurrent programs, indicating the main principles behind compositionality. Having achieved a compositional framework based on classical Hoare triples, we discuss extensions to achieve a convenient formalism to specify and verify reactive systems that have an intensive interaction with their environment. Next this Hoare-style framework is adapted to specify and verify real-time properties, and a compositional proof method is formulated for real-time distributed computing. Compositional reasoning during top-down development of a real-time program is illustrated by an example concerning a watchdog timer.

Keywords. Compositional methods for concurrency; real-time applications; distributed computing.

1. Introduction

Formal methods for the specification and verification of distributed systems can be classified from the viewpoint of expressibility (which properties can be specified), specification language (e.g., temporal logic, Hoare triples and first-order assertions), and programming features (such as time-out, various communication mechanisms and concurrency). In this paper we concentrate on the distinction between proof methods that are only applicable to complete program code and methods that can be used to verify design steps during the process of program development. We sketch the development from a-posteriori methods (requiring the complete program text) towards compositional methods (supporting verify-while-design). Compositionality can be considered as a requirement for hierarchical, structured, program derivation. A separation of concerns is desired between the use of (and the reasoning about) a module and its implementation. This leads to the following definition of *compositionality* for proof methods:

> Properties of a compound programming language construct (such as sequential composition and parallel composition) can be deduced from specifications for its constituent parts without any further information about the internal structure of these parts.

In general, compositional program specification and verification dictates, as a principle, that all aspects of program execution which are required to define the meaning of a compound statement from its constituents, must be explicitly addressed in semantics and assertion language alike. In *semantics* because, otherwise, no compositional semantics can be defined, since compositionality in semantics requires that the meaning of a compound statement is a function of the meaning of its parts (the guiding principle of denotational semantics). In *specification languages* because, otherwise, no compositional verification rules can be formulated in which the specification of a compound statement should follow from specifications of its constituent parts without knowledge about their internal structure (the internal structure often providing implicit information which has not been explicitly stated in the specification, but is used in non-compositional methods (Owicki & Gries 1976; Apt *et al* 1980)). The rationale for this principle is that one must be able to specify the behaviour of a module in isolation, i.e. without any implicit prior assumption regarding the environment within which it ultimately functions. Hence, all assumptions which are needed regarding the environment–because these influence the behaviour of a module–must be made explicit as parameters (in the semantics and specification of that module alike) for only then one can abstract away from the remaining aspects (such as inner syntactic structure).

In case of shared variable communication this compositionality principle implies that when defining the behaviour of a module any change of a shared variable by the environment must be explicitly expressed as an assumption of that module regarding its environment. This is worked out in Aczel's model for shared variable semantics as cited in de Roever (1985b, pp. 181–207). Similarly, when considering distributed communication via input/output-statements, the specification of, e.g., an input statement in one module requires explicit expressibility of assumptions regarding a corresponding output statement in another module. In case one abstracts away from blocking behaviour only assumptions regarding the value communicated must be expressible. If blocking behaviour is a focus of interest, this is again an assumption regarding program execution which must be stated explicitly; i.e. one has to state the effect of no communication partner being available in the assertion language and one must be able to express the assumption that no partner is available in the assertion language.

In this paper we also discuss the compositional verification of real-time properties for distributed systems. When the timing behaviour of a statement is considered, all factors concerning the execution of this statement which influence that timing behaviour must be expressible. For example, for real-time systems we use in this paper the maximal progress assumption with respect to distributed i/o-communication from Koymans *et al* (1988): no input or output statement should wait for communication when its partner is also ready to communicate. This aspect of timing behaviour requires, indeed, that one must be able to express when a partner is waiting to communicate. For, otherwise, maximal progress would not be expressible within the semantics, and hence timing behaviour of i/o-statements could not be characterized. This maximal progress assumption, which represents the situation that each process has its own processor, can be generalized to multiprogramming where several processes may share a single processor. By introducing priorities for processes on a single processor, certain statements which are ready to execute will not be executed on account of their priority and because at most one action can be executed at a time on a uniprocessor. Modelling the timing behaviour of such statements requires that the semantics, and hence the specification language, contains primitives to state explicitly when a statement is executing and when it is requesting processor time with a certain priority. The semantic aspects of reasoning formally about real-time and scheduling by means of priorities are addressed technically in Hooman (1991a).

This paper is structured as follows. A programming language with synchronous message passing is defined in § 2. Section 3 contains a description of a classical non-compositional method, and we indicate how a compositional proof system can be achieved for Hoare triples (pre-condition, program, post-condition). For the specification and verification of reactive systems, these triples are extended in § 4 with assertions (called assumption and commitment) that specify the communication interface between a program and its environment. In § 5 we adapt this Hoare-style framework to specify real-time properties of programs, and we give the details of a compositional proof system for real-time distributed systems. The formalism of § 5 is illustrated by an example of a watchdog timer in § 6. The extension of this formalism to assumption/commitment based reasoning for real-time is described in § 7. In § 8 we sketch the development of the field, leading to a description of the state of the art and the place of our work therein.

2. Syntax

We give syntax and informal semantics of a programming language for distributed synchronous message-passing. Our language is akin to Occam (1988) with concurrent processes that communicate via message passing along unidirectional channels, each connecting two processes. Communication is synchronous, i.e., both the sender and the receiver have to wait until a communication partner is available.

Let $CHAN$ be a nonempty set of channel names, VAR be a nonempty set of program variables, and VAL be a denumerable domain of values. $\mathbb{N}$ denotes the set of natural numbers (including 0). The syntax of our programming language is given in table 1, with $n \in \mathbb{N}$, $n \geqslant 1$, $c, c_1, \ldots, c_n \in CHAN$, $x, x_1, \ldots, x_n \in VAR$, and $\vartheta \in VAL$.

Table 1. Syntax programming language.

Expression	$e ::= \vartheta \mid x \mid e_1 + e_2 \mid e_1 + e_2 \mid e_1 \times e_2$
Boolean expression	$b ::= e_1 = e_2 \mid e_1 < e_2 \mid \neg b \mid b_1 \vee b_2$
Statement	$S ::= x := e \mid c!e \mid c?x \mid S_1 ; S_2 \mid G \mid \star G \mid S_1 \parallel S_2$
Guarded command	$G ::= [\square_{i=1}^{n} b_i \rightarrow S_i] \mid [\square_{i=1}^{n} b_i ;\ c_i?x_i \rightarrow S_i]$

Informally, the statements of our programming language have the following meaning.

Atomic statements

- Assignment $x := e$ assigns the value of expression e to the variable x.
- Output statement $c!e$ is used to send the value of expression e on channel c as soon as an input command $c?x$ is available. Since we assume synchronous communication, such an output statement is suspended until a parallel process executes a corresponding input statement.
- Input statement $c?x$ is used to receive a value via channel c and assign this value to the variable x. As for the output command, such an input statement has to wait for a corresponding partner before a (synchronous) communication can take place.

Henceforth we will often refer to an input or output statement as an *i/o-statement.*

Compound statements

- $S_1; S_2$ indicates sequential composition: first execute S_1, and continue with the execution of S_2 if and when S_1 terminates.
- Guarded command $[\Box_{i=1}^{n} b_i \rightarrow S_i]$. If none of the b_i evaluate to true then this guarded command terminates after evaluation of the booleans. Otherwise, non-deterministically select one of the b_i that evaluates to true and execute the corresponding statement S_i.
- Guarded command $[\Box_{i=1}^{n} b_i;\ c_i?x_i \rightarrow S_i]$. A guard (the part before the arrow) is *open* if its boolean part evaluates to true. If none of the guards is open, the guarded command terminates after evaluation of the booleans. Otherwise, wait until the communication of one of the open guards can be performed and continue with the corresponding S_i.
- Iteration $*G$ indicates repeated execution of guarded command G as long as at least one of the guards is open. When none of the guards is open $*G$ terminates.
- $S_1 \| S_2$ indicates parallel execution of the statements S_1 and S_2. The components S_1 and S_2 of a parallel composition are often called *processes.*

Henceforth we use $\equiv$ to denote syntactic equality. Conventional abbreviations are used, such as $true \equiv 0 = 0$, $false \equiv \neg\, true$, $b_1 \wedge b_2 \equiv \neg(\neg b_1 \vee \neg b_2)$ etc.

For a guarded command $G \equiv [\Box_{i=1}^{n} b_i \rightarrow S_i]$ or $G \equiv [\Box_{i=1}^{n} b_i;\ c_i?x_i \rightarrow S_i]$, we define $b_G \equiv b_1 \vee \ldots \vee b_n$. Observe that conventional programming constructs can be defined as an abbreviation:

$$\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi} \equiv [b \rightarrow S_1 \Box \neg b \rightarrow S_2] \text{ and } \textbf{while } b \textbf{ do } S \textbf{ od} \equiv *[b \rightarrow S].$$

3. Compositionality

In § 3.1 we explain the principles of traditional non-compositional methods. The development towards compositional proof systems based on Hoare triples is described in § 3.2.

3.1 *Non-compositional methods*

Classical verification methods for parallel processes, such as Owicki & Gries (1976) for shared variable communication and Apt *et al* (1980), Levin & Gries (1981) for synchronous message passing, consist of two stages. First a *local* correctness proof is given for each of the sequential process by associating assertions with locations in the program. In the second, *global*, stage a consistency check is applied to the local proofs.

- For shared variables this is the *interference freedom* test which verifies that assertions in the proof of one process remain valid under actions of other processes.
- For communication via message passing the *cooperation* test is applied to verify correctness of assertions attached to locations after input- and output-statements.

Such methods are not compositional because at parallel composition they require the complete program text, annotated with assertions, of the constituent processes. Moreover, they are only suited for top-level parallelism, that is, to prove correctness of programs of the form $S_1 \| \ldots \| S_n$ where $S_1, \ldots, S_n$ are sequential processes.

As an example, we consider in more detail the method of Apt *et al* (1980) for synchronous message passing. This method is based on *Hoare triples* (Hoare 1969) that is, on correctness formulae of the form $\{p\}\ S\ \{q\}$ which have the following meaning: if we start program S in a state satisfying assertion p (the pre-condition) and if program S terminates then assertion q (the post-condition) holds for the termination state. For example, $\{x = 5\} x := x + 1 \{x = 6\}$ is a valid Hoare triple.

First we indicate how a proof system can be formulated in which valid Hoare triples can be derived for sequential programs. Let $q[e/x]$ denote the textual substitution of expression e for each free occurrence of variable x in assertion q. Then we have the following assignment axiom:

Axiom 3.1. (Assignment)

$$\{q[e/x]\} x := e \{q\}.$$

Example 3.1. With this axiom we can derive $\{x = 5\} x := x + 1 \{x = 6\}$, because $(x = 6)[x + 1/x]$ equals $x + 1 = 6$, which is equivalent to $x = 5$.

Furthermore the proof system contains rules for compound constructs. For instance, sequential composition is modelled by the following rule:

Rule 3.2. (Sequential composition)

$$\frac{\{p\} S_1 \{r\},\ \{r\} S_2 \{q\}}{\{p\} S_1;\ S_2 \{q\}}.$$

By such a rule the formula below the line can be derived from the formulae above the line. Soundness of the rule is proved by showing that validity of the formulae above the line implies that the formula below the line is valid. Note that this rule is compositional because the formula for $S_1;\ S_2$ is derived without using the structure of S_1 or S_2. To strengthen pre-conditions and weaken post-conditions, the proof system contains the following rule:

Rule 3.3. (Consequence)

$$\frac{p \rightarrow p_1,\ \{q_1\} S \{q_1\},\ q_1 \rightarrow q}{\{p\} S \{q\}}.$$

To illustrate the rule for parallel composition in Apt *et al* (1980), we consider the proof of

$$\{y=3\}\ (a?x;\ x:=x+1;\ b!(x+2))\,\|\,(a!y;\ b?y;\ y:=y+2)\{x=4 \wedge y=8\}.$$

In the first stage we attach assertions to all locations in the program text of the two processes, leading to so-called *proof outlines*:

$$\{\text{true}\}a?x\{x=3\};\ x:=x+1\{x=4\};\ b!(x+2)\{x=4\},$$

and

$$\{y=3\}a!y\{y=3\};\ b?y\{y=6\};\ y:=y+2\{y=8\}.$$

In this stage only the post-conditions of assignments are verified: from the assignment axiom we obtain $\{x=3\}x:=x+1\{x=4\}$ and $\{y=6\}y:=y+2\{y=8\}$. Observe that the post-conditions of the input statements $a?x$ and $b?y$ express assumptions about the values sent by the communication partner.

These assumptions are verified in the second stage by means of the *cooperation* test.[1] In general, this test requires that for $\{p_1\}c?x\{q_1\}$ and $\{p_2\}\ c!e\{q_2\}$ in the proof outlines of two processes we have to prove $\{p_1 \wedge p_2\}c?x\|c!e\{q_1 \wedge q_2\}$, which is equivalent to proving $\{p_1 \wedge p_2\}x:=e\{q_1 \wedge q_2\}$. In our example this leads to the proof obligations:

$$\{true \wedge y=3\}a?x\,\|\,a!y\{x=3 \wedge y=3\}$$

and

$\{y=3 \wedge x=4\}b?y\,\|\,b!(x+2)\{y=6 \wedge x=4\}$ which are easy to prove.

After the verification of the first two stages we obtain the conjunction of all pre-conditions from the sequential processes as the pre-condition of the complete program and the conjunction of the post-conditions as the final post-condition. In our example this leads to the pre-condition $true \wedge y=3$ and the post-condition $x=4 \wedge y=8$ which are equivalent to the required conditions.

3.2 *Towards compositionality*

In this section we discuss how a compositional proof method can be obtained for programs which communicate via synchronous message passing. First the cooperation test from Apt *et al* (1980) is removed by not allowing implicit assumptions in the post-conditions of i/o-statements. The local proof of a sequential program should be valid in any arbitrary environment. Since this would weaken the method, and any valid post-condition should be provable, we use a *history variable h* which denotes the communication history of the complete program. A (communication) *history* is a sequence of records (c, ϑ) where c is a channel name and ϑ a value. For example, $\langle(c\,5), (b,6), (a,8), (b,0)\rangle$ is a history expressing four communications: first one via channel c with value 5, then a communication via b with value 6, etc. Let $\langle\rangle$ denote theempty sequence. History variable h does not appear in the program,but it is updated implicitly in the semantics of i/o-statements. This leads to the following valid formulae.

- For an output command we have, for example,

$$\{h=\langle(c,5)\rangle\}b!6\{h=\langle(c,5),(b,6)\rangle\}.$$

[1] In the full method of Apt *et al* (1980) auxiliary variables and a global invariant are used to make the method complete, i.e., to guarantee that any valid Hoare triple can be proved.

- For an input statement $c?x$ we can only express in the post-condition that there exists a value which is communicated via c and assigned to x. For instance,

$$\{h=\langle\rangle\}c?x\{\exists v{:}h=\langle(c,v)\rangle \wedge x=v\}.$$

Example 3.2. To prove $(true\}$ $c!5 \| c?x\{x=5\}$, we use the Hoare triples

$$\{h=\langle\rangle\}c?5\{h=\langle(c,5)\rangle\} \text{ and } \{h=\langle\rangle\}c?x\{\exists v{:}h=\langle(c,v)\rangle \wedge x=v\}.$$

Suppose the pre- and post-conditions after parallel composition are obtained by simply taking the conjunction of, respectively, pre- and post-conditions of the sequential programs. Then

$$\{h=\langle\rangle\}c!5 \| c?x\{h=\langle(c,5)\rangle \wedge \exists v{:}h=\langle(c,v)\rangle \wedge x=v\}.$$

Since the post-condition implies $\exists v{:}v=5 \wedge x=v$, and hence $x=5$, the consequence rule leads to $\{h=\langle\rangle\}c!5 \| c?x\{x=5\}$. By a so-called *substitution rule* (not given in this paper), we could substitute $\langle\rangle$ for h in the pre-condition, thus obtaining pre-condition *true.* □

Example 3.2 suggests the following rule:

$$\frac{\{p_1\}S_1\{q_1\},\ \{p_2\}S_2\{q_2\}}{\{p_1 \wedge p_2\}S_1 \| S_2\{q_1 \wedge q_2\}}.$$

Example 3.3. Consider again $S_1 \| S_2$, where $S_1 \equiv a?x;\ x{:}=x+1;\ b!(x+2)$ and $S_2 \equiv a!y;\ b?y;\ y{:}=y+2$. First derive the following Hoare triples:

$$\{h=\langle\rangle\}a?x\{\exists v_1{:}h=\langle(a,v_1)\rangle \wedge x=v_1\},$$

$$\{\exists v_1{:}h=\langle(a,v_1)\rangle \wedge x=v_1\}x{:}=x+1\{\exists v_1{:}h=\langle(a,v_1)\rangle \wedge x=v_1+1\},$$

and

$$\{\exists v_1{:}h=\langle(a,v_1)\rangle \wedge x=v_1+1\}b!(x+2)$$
$$\{\exists v_1{:}h=\langle(a,v_1),\ (b,v_1+3)\rangle \wedge x=v_1+1\}.$$

By two applications of the sequential composition rule we obtain

$$\{h=\langle\rangle\}a?x;\ x{:}=x+1;\ b!(x+2)$$
$$\{\exists v_1{:}h=\langle(a,v_1),\ (b,v_1+3)\rangle \wedge x=v_1+1\}.$$

Similarly,

$$\{h=\langle\rangle \wedge y=3\}a!y;\ b?y;\ y{:}=y+2\{\exists v_2{:}h=\langle(a,3),\ (b,v_2)\rangle \wedge y=v_2+2\}.$$

Then the parallel composition rule above leads to

$$\{h=\langle\rangle \wedge y=3\}S_1 \| S_2\{\exists v_1{:}h=\langle(a,v_1),\ (b,v_1+3)\rangle \wedge x=v_1+1 \wedge$$
$$\exists v_2{:}h=\langle(a,3),\ (b,v_2)\rangle \wedge y=v_2+2\}.$$

The post-condition implies $\exists v_1,v_2{:}v_1=3 \wedge v_2=v_1+3 \wedge x=v_1+1 \wedge y=v_2+2$, which leads to $x=4 \wedge y=8$. Thus, by the consequence rule. $\{h=\langle\rangle \wedge y=3\}S_1 \| S_2$ $\{x=4 \wedge y=8\}$.
(Again $h=\langle\rangle$ in the pre-condition can be removed by a substitution rule.) □

Although this works nicely for two processes, the next example shows that there is a problem if more than two processes are involved.

Example 3.4. Consider $S_1 \| S_2 \| S_3$, where $S_1 \equiv a!0;\ b?x$, $S_2 \equiv a?y;\ c!(y+1)$, and $S_3 \equiv c?z;\ b!(z+1)$. Similar to example 3.3, we could first prove

$$\{h = \langle\rangle\} S_1 \{q_1 \equiv \exists v_1 : h = \langle (a,0), (b,v_1) \rangle \wedge x = v_1\}$$

and

$$\{h = \langle\rangle\} S_2 \{q_2 \equiv \exists v_2 : h = \langle (a,v_2), (c,v_2+1) \rangle \wedge y = v_2\}.$$

But then the conjunction of q_1 and q_2 implies *false* whereas $S_1 \| S_2 \| S_3$ terminates and hence does not satisfy post-condition *false*.

The problem is that h denotes the global history of the complete program – e.g., consisting of three processes – whereas each of the processes in isolation can only describe the history on its own channels. A possible solution is to give each process its own history variable, and to combine these local history variables at parallel composition. This is done in Soundararajan (1984a), using a predicate *compat*. Zwiers (Zwiers *et al* 1984; Zwiers 1989), however, shows that a concise and simple rule for parallel composition can be formulated if each process uses projections of global history variable h onto its own channels. Such a projection expresses the view of a particular process on the global history. Formally, the *projection* of h onto a set of channel names *cset*, notation h_{cset}, denotes the sequence obtained from the history denoted by h by removing all records with a channel name not in *cset*. For instance, if $h = \langle (a,0), (c,1), (b,3) \rangle$ then $h_{\{c\}} = \langle (c,1) \rangle$, $h_{\{b,c\}} = \langle (c,1), (b,3) \rangle$, and $h_{\{d\}} = \langle\rangle$. Henceforth we write h_c, h_{bc}, and h_d instead of $h_{\{c\}}$, $h_{\{b,c\}}$, and $h_{\{d\}}$, respectively.

In the rule for $S_1 \| S_2$ we require that the post-condition of S_i only refer to history h via projections on channels occurring in S_i, for $i = 1,2$. If, moreover, the post-condition of S_i only refers to program variables of S_i, then the following rule for parallel composition is sound;

Rule 3.4. (Parallel composition)

$$\frac{\{p_1\} S_1 \{q_1\},\ \{p_2\} S_2 \{q_2\}}{\{p_1 \wedge p_2\} S_1 \| S_2 \{q_1 \wedge q_2\}}.$$

Observe that this is a compositional rule because a Hoare triple for $S_1 \| S_2$ can be derived without knowing the internal structure of S_1 and S_2. To obtain a valid rule we only impose a simple syntactic requirement on assertions and processes (the post-condition of a process should only refer to channels and variables of the process itself). This simple syntactic check replaces the cooperation test which requires proof outlines for the complete program text of the processes.

Except for bottom-up verification such a compositional rule can be used for top-down development. Therefore, a triple $\{p\} S \{q\}$ is considered as a specification for a program S. Suppose we decide to implement S as $S_1 \| S_2$. If we can find assertions p_1, q_1, p_2, and q_2 that satisfy $p \rightarrow p_1 \wedge p_2$ and $q_1 \wedge q_2 \rightarrow q$, and moreover certain syntactic requirements on the post-conditions hold, then S_1 and S_2 can be implemented independently, using specifications $\{p_1\} S_1 \{q_1\}$ and $\{p_2\} S_2 \{q_2\}$.

Since in this compositional framework programs can be considered as black boxes, and verification is done on the basis of the specifications only, we can allow nested parallelism in programs as expressed by the syntax of § 2.

4. Extensions of Hoare triples

A Hoare triple is perfectly suited to describe the observable behaviour of a sequential program which is given by initial and final states. For a parallel program also the communication behaviour on its external channels is observable. Hence a specification of a parallel component should express this communication interface. Note, however, that a specification $\{p\}S\{q\}$ has an important limitation: it only specifies the behaviour of S if S terminates. All non-terminating computations of S satisfy such a specification trivially. Thus the post-condition cannot be used to express the communication interface of non-terminating programs. Therefore, a Hoare triple is extended with an invariant, called *commitment* in this paper, which must hold throughout the computation. This leads to a formula of the form C: $\{p\}S\{q\}$ where commitment C describes the communication interface of S during its execution. The success of such formulae in many applications is based on a simple rule for parallel composition in which, besides conjunctions for pre- and post-conditions, also the conjunction of commitments can be taken.

Rule 4.1. (Parallel composition)

$$\frac{C_1{:}\{p_1\}S_1\{q_1\},\ C_2{:}\{p_2\}S_2\{q_2\}}{C_1 \wedge C_2{:}\{p_1 \wedge p_2\}S_1 \| S_2\{q_1 \wedge q_2\}}$$

provided, for $i = 1,2$, the assertions C_i and q_i refer to h only via projections on the channels occurring in S_i, and q_i only refers to program variables of S_i.

In this formalism the influence of the environment on the communication behaviour of a program can be expressed by using implications in the commitment. The next example indicates that for so-called reactive systems (Harel & Pnueli 1985) which have an intensive interaction with their environment, this style of specification often leads to proofs using inductive arguments. This motivates a final extension of the Hoare-style formulae.

In the following examples, $seq_1 \preccurlyeq seq_2$ expresses that sequence seq_1 is an initial prefix of sequence seq_2. For a history variable h, a set of channels *cset*, and a number i, we use $h_{cset}[i] = (c, \vartheta)$ to denote that the record (c, ϑ) is the ith element of the sequence denoted by h_{cset}. We assume that $h_{cset}[i] = (c, \vartheta)$ is true if i is greater than the length of h_{cset}.

Example 4.1. We verify two reactive, non-terminating, processes that have a close interation. Consider $S_1 \equiv c!1; \star[d?x \rightarrow c!(x+1)]$ and $S_2 \equiv \star[c?y \rightarrow d!(y+1)]$. The aim is to prove that $S_1 \| S_2$ satisfies the commitment $h_{cd} \preccurlyeq \langle (c,1), (d,2), (c,3), (d,4), (c,5), \ldots \rangle$, that is,

$$\forall i \geqslant 1{:}h_{cd}[2i-1] = (c, 2i-1) \wedge h_{cd}[2i] = (d, 2i).$$

Define

$$C_1 \equiv h_{cd}[1] = (c,1) \wedge (\forall i \geqslant 1 \forall v{:}h_{cd}[2i] = (d,v) \rightarrow h_{cd}[2i+1] = (c, v+1)).$$

Then for S_1 we can prove

$$C_1{:}\{h_{cd} = \langle\rangle\}c!1;\star[d?x \rightarrow c!(x+1)]\{\mathit{false}\}.$$

Similarly, for S_2 we define

$$C_2 \equiv (\forall i \geqslant 1 \forall v{:}h_{cd}[2i-1] = (c,v) \rightarrow h_{cd}[2i] = (d, v+1)).$$

Then we have

$$C_2:\{h_{cd}=\langle\rangle\}\star[c?y\rightarrow d!(y+1)]\{false\}.$$

Since the required syntactic conditions are fulfilled, we can apply the rule for parallel composition which leads to

$$C_1\wedge C_2:\{h_{cd}=\langle\rangle\}S_1\,\|\,S_2\{false\}.$$

To prove that $C_1\wedge C_2$ implies the required commitment, we prove by induction on i that, for $i\geqslant 1$, $h_{cd}[2i-1]=(c,2i-1)\wedge h_{cd}[2i]=(d,2i)$.

- Basic step. For $i=1$ we have, by C_1, that $h_{cd}[1]=(c,1)$ and thus from C_2 we obtain $h_{cd}[2]=(d,2)$.
- Induction step. Assume $h_{cd}[2i-1]=(c,2i-1)\wedge h_{cd}[2i]=(d,2i)$. Then $h_{cd}[2i]=(d,2i)$ implies, by C_1, that $h_{cd}[2i+1]=(c,2i+1)$, and thus $h_{cd}[2(i+1)-1]=(c,2(i+1)-1)$. Using C_2 this leads to $h_{cd}[2(i+1)]=(d,2(i+1))$. □

This example illustrates that assumptions about the environment are important in the specification of a process, and it indicates that mutual assumptions of processes about each others communication interface usually leads to correctness proofs using inductive reasoning. Based on these observations we present an extension of the correctness formulae in which a process can be specified relative to explicit assumptions about its environment, and an inductive relation is incorporated in the specification. Therefore the specification formula is extended with a second invariant, called *assumption*, which expresses assumptions about the environment and by which we can strengthen post-condition and commitment. This leads to formulae of the form $(A,C):\{p\}S\{q\}$, where
A is an *assumption* describing the expected behaviour of the environment of S, and
C is a *commitment* which is guaranteed by process S itself, as long as the environment does not violate the assumption.

The general idea is that assumption and commitment reflect the communication interface between parallel components (and hence do not contain program variables), whereas pre- and post-condition facilitate the reasoning at sequential composition.

Example 4.2. In this formalism assumptions about the values sent by the environment can be expressed explicitly. For instance, the assertion $h_c\preccurlyeq\langle(c,3)\rangle$ can be used as an assumption:

$$(h_c\preccurlyeq\langle(c,3)\rangle,\ true):\{true\}c?x\{x=3\}.$$

This assumption expresses that if a communication along c takes place then the environment will send the value 3. The next formula shows that it can be used for a commitment about the next communication:

$$(h_c\preccurlyeq\langle(c,3)\rangle,\ h_a\preccurlyeq\langle(a,4)\rangle:\{true\}c?x;\ a!(x+1)\{x=3\}.$$ □

Assumption/commitment based reasoning was introduced in Misra & Chandy (1981). A proof system for these formulae has been given in Zwiers *et al* (1984). In this paper we discuss the proof obligations for assumptions and commitments in the parallel composition rule. Consider the parallel composition $S_1\,\|\,S_2$, and suppose we have assumption-commitment pairs (A_1,C_1) for S_1 and $(A_2;C_2)$ for S_2. Which conditions have to be verified to obtain a pair (A, C) for $S_1 \| S_2$? Consider assumption A_2 of S_2:

- A_2 may contain assumptions about joint channels of S_1 and S_2 which connect these two processes; these assumptions must be justified by the commitment C_1 of S_1.
- A_2 may contain assumptions about external channels of S_2. These assumptions are maintained in the new network assumption A for $S_1 \| S_2$.

This leads to the following proof obligation: $A \wedge C_1 \rightarrow A_2$. Similarly, $A \wedge C_2 \rightarrow A_1$.

To obtain a sound rule with these implications, the meaning of a formula (A_i, C_i): $\{p_i\} S_i \{q_i\}$ has to be defined carefully. A simple implication between A_i and C_i would with the implications above and $A \equiv \mathit{true}$ lead to circular reasoning, that is, $A_1 \rightarrow C_1 \rightarrow A_2 \rightarrow C_2 \rightarrow A_1$. Therefore in defining the meaning of (A_i, C_i): $\{p_i\} S_i \{q_i\}$ we require that if p_i holds in the initial state then (1) C_i holds initially, and (2) C_i holds after every communication provided A_i holds after all preceding communications. This inductive step inside the meaning of formulae is sufficient to avoid circularity (see Misra & Chandy 1981, Zwiers *et al* 1984).

As in rule 4.1 we can take the conjunction of pre-conditions, post-conditions, and commitments, provided, for $i = 1,2$, the assertions A_i, C_i, p_i and q_i of S_i refer only to h via projections on the channels of S_i, and p_i and q_i only refer to program variables of S_i. (Program variables are not allowed in A_i and C_i.) With these constraints, the following rule for parallel composition is valid.

Rule 4.2. (Parallel compositon A–C)

$$\frac{\begin{array}{c}(A_1, C_1)\colon \{p_1\} S_1 \{q_1\},\ (A_2, C_2)\colon \{p_2\} S_2 \{q_2\} \\ A \wedge C_1 \rightarrow A_2,\ A \wedge C_2 \rightarrow A_1\end{array}}{(A, C_1 \wedge C_2)\colon \{p_1 \wedge p_2\} S_1 \| S_2 \{q_1 \wedge q_2\}}.$$

Example 4.3. Consider $S_1 \| S_2$ where $S_1 \equiv a?x;\ x := x + 1;\ b!(x + 2)$ and $S_2 \equiv c?y;\ a!y;\ b?y;\ y := y + 2$. Then for S_1 and S_2 we can derive

$$(A_1 \equiv h_a \preccurlyeq \langle (a,3) \rangle,\ C_1 \equiv h_b \preccurlyeq \langle (b,6) \rangle\colon \{h_{ab} = \langle\rangle\} S_1 \{x = 4\},$$

and

$$(A_2 \equiv h_c \preccurlyeq \langle (c,3) \rangle \wedge h_b \preccurlyeq \langle (b,6) \rangle,\ C_2 \equiv h_a \preccurlyeq \langle (a,3) \rangle)\colon \{h_{abc} = \langle\rangle\} S_2 \{y = 8\}.$$

Since S_1 and S_2 communicate with each other via the channels a and b, the remaining assumption for $S_1 \| S_2$ concerns the external channel c: $A \equiv h_c \preccurlyeq \langle (c,3) \rangle$.

Then $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$, thus the parallel composition rule leads to

$$(A, C_1 \wedge C_2)\colon \{h_{abc} = \langle\rangle\} S_1 \| S_2 \{x = 4 \wedge y = 8\}. \qquad \square$$

Example 4.4. Consider again the two reactive processes from example 4.1:

$$S_1 \equiv c!1; \star[d?x \rightarrow c!(x + 1)] \text{ and } S_2 \equiv \star[c?y \rightarrow d!(y + 1)].$$

We show how the assumption/commitment formalism can be used to prove for $S_1 \| S_2$ the commitment $\forall i \geqslant 1\colon h_{cd}[2i - 1] = (c, 2i - 1) \wedge h_{cd}[2i] = (d, 2i)$.
Define

$$A_1 \equiv \forall i \geqslant 1\colon h_{cd}[2i] = (d, 2i)$$

and

$$C_1 \equiv \forall i \geqslant 1\colon h_{cd}[2i - 1] = (c, 2i - 1).$$

Then for S_1 we can prove

$$(A_1, C_1):\{h_{cd} = \langle\rangle\}c!1;\ \star[d?x \rightarrow c!(x+1)]\{false\}.$$

Similarly, for S_2 we define

$$A_2 \equiv \forall i \geqslant 1: h_{cd}[2i-1] = (c, 2i-1),$$

and

$$C_2 \equiv \forall i \geqslant 1: h_{cd}[2i] = (d, 2i).$$

Then we have

$$(A_2, C_2):\{h_{cd} = \langle\rangle\}\star[c?y \rightarrow d!(y+1)]\{false\}.$$

Let $A \equiv true$. Then $A \wedge C_2 \rightarrow A_1$ and $A \wedge C_1 \rightarrow A_2$. Since the other conditions on the assertions are also satisfied, we can apply the rule for parallel composition, leading to

$$(true,\ C_1 \wedge C_2):\{h_{cd} = \langle\rangle\}S_1 \| S_2\{false\}.$$

Clearly $C_1 \wedge C_2$ is equivalent to the required commitment. □

Observe that in the correctness proof for the two reactive processes from example 4.4 there is no explicit inductive argument. The requirement inductive reasoning is performed only once in the soundness proof of the parallel composition rule (rule 4.2). In this respect we have obtained a rule for parallel composition which is the analogue of Hoare's while rule for sequential programs (Hoare 1969).

5. Compositionality and real-time

In this chapter we adapt the compositional Hoare-style proof systems from the previous chapters to real-time. We describe in detail a compositional method to specify and verify timing constraints. By describing the details of a particular compositional proof method, we illustrate the general outline of such a description which should consist of the following points.

1. A description of the programming language, i.e., syntax and informal semantics.
2. A formal semantics of the programming language.
3. The definition of an assertion language in which properties of programs can be expressed. For this assertion language we also have to give syntax, informal meaning, and formal interpretation.
4. The definition of a correctness formula that relates programs and assertions. Using the semantics of the programming language and the interpretation of assertions, the validity of such a correctness formula can be defined formally.
5. A proof system in which, by rules and axioms, correctness formulae can be derived formally.
6. The proof of soundness and (relative) completeness of the proof system: show that every correctness formula that can be derived is also valid, and that every valid formula can be derived (assuming that valid assertions can be derived).

As an example, we consider in this section a compositional proof method for distributed real-time systems based on Hooman (1991b). Concerning the points above, we describe:

1. A real-time programming language with nested parallelism, communication via synchronous message passing, and time-outs.
2. The semantic model which is used to give a denotational semantics for the programming language; the meaning of a program is given by a set of models where each model describes a possible computation of the program.
3. A first-order assertion language which includes primitives to specify the timing behaviour of programs.
4. A correctness formula of the form $C{:}\{p\}S\{q\}$.
5. A compositional proof system to derive these extended Hoare triples.
6. The proofs of soundness and (relative) completeness for the proof system given in this section are not given here. The reader is referred to Hooman (1991b) for all details about these proofs.

The syntax and informal semantics of our real-time programming language are given in §5.1. A semantic model for this language is described in §5.2. In §5.3 we define the syntax and the interpretation of assertion language and correctness formulae. A compositional proof system for this formalism is presented in §5.4.

5.1 *Real-time programming language*

Our real-time programming language is based on the Occam-like language from §2 and is akin to real-time versions of CSP as defined in Koymans *et al* (1988) and Huizing *et al* (1987). We add a real-time statement **delay** *d* which suspends the execution for (at least) *d* time units. This statement is also used in the language Ada (Ada 1983) and corresponds to a *wait d* statement (Koymans *et al* 1988; Huizing *et al* 1987). Similar to a delay-statement in select construct of Ada, such a delay-statement is allowed in a guard of a guarded command to enable the programming of time-outs.

To investigate the basic real-time framework and to highlight the main points, no program variables are used – we consider only the (real-time) communication behaviour. In Hooman (1991b) we show that this framework can be extended to a language with program variables. Processes communicate and synchronize by message passing via unidirectional channels, each connecting two processes. Communication is synchronous.

5.1a *Syntax and informal meaning*: Let $TIME$ be some countable ordered time domain and ∞ a special symbol, $\infty \notin TIME$. The syntax of our programming language is given in table 2, with $n \in \mathbb{N}$, $c, c_1, \ldots, c_n \in CHAN$, $d \in TIME$, and $d_0 \in TIME \cup \{\infty\}$, $d_0 > 0$.

Table 2. Syntax programming language.

Statement	$S ::= \mathbf{skip} \mid \mathbf{delay}\ d \mid c! \mid c? \mid$ $S_1 ; S_2 \mid G \mid \star G \mid S_1 \parallel S_2$
Guarded command	$G ::= [\square_{i=1}^{n} c_i? \rightarrow S_i \square\, \mathbf{delay}\ d_0 \rightarrow S]$

Since we have slightly modified the syntax of our programming language, we briefly mention the informal meaning of statements.

Atomic statements

- **skip** terminates immediately.
- **delay** d suspends execution for d time units.
- $c!$ is used to send a signal along channel c. (In this chapter we do not consider the value transmitted.) Since we are assuming synchronous communication, a statement $c!$ is suspended until the receiving process executes a statement $c?$.
- $c?$ is used to receive a signal along channel c. An input statement $c?$ is suspended until the sending process executes an output statement $c!$.

Compound statements

- $S_1; S_2$ indicates sequential composition of statements S_1 and S_2.
- Guarded command $[\Box_{i=1}^{n} c_i? \to S_i \Box \mathbf{delay}\ d \to S]$ is executed as follows: wait at most d time units for some input guard $c_i?$ to become enabled, that is, until communication can actually occur along one of the c_i because a communication partner becomes available. If at least one of the c_i-communications is possible (before d time units have elapsed), one of these communications (non-deterministically chosen) is performed and thereafter the corresponding S_i is executed. If $d \neq \infty$ and no guard is enabled within d time units after the start of the execution of the command, then S is executed.

Example 5.1. This construct makes it possible to model a *time-out*, i.e., to restrict the waiting period for certain communications. Consider the guarded command $[c?x \to S_1 \Box \mathbf{delay}\, 5 \to S_2]$; if there is no partner available for the input statement within 5 time units then the delay-alternative is taken and S_2 is executed. □

- $\star G$ indicates repeated execution of guarded command G. Since we do not consider boolean guards in this chapter, execution of $\star G$ never terminates.
- $S_1 \| S_2$ indicates parallel execution of S_1 and S_2.

We often use $[\Box_{i=1}^{n} c_i? \to S_i]$ as an abbreviation of $[\Box_{i=1}^{n} c_i? \to S_i \Box \mathbf{delay}\ \infty \to S]$, if $n > 0$. Let *DCHAN* be the set of channels extended with directional channels;

$$DCHAN = CHAN \cup \{c! \mid c \in CHAN\} \cup \{c? \mid c \in CHAN\}.$$

DEFINITION 5.1 (Channels occurring in statement)

The set of (directional) channels occurring in a statement S, notation $dch(S)$, is defined as the smallest subset of *DCHAN* such that if c is an output channel of S then $\{c, c!\} \subseteq dch(S)$, and if c is an input channel of S then $\{c, c?\} \subseteq dch(S)$.

5.1b *Syntactic restrictions*: A number of syntactic constraints are imposed upon statements to guarantee that a channel connects exactly two processes. With the definition of $dch(S)$ above we can express these restrictions formally as follows:

- For $S_1; S_2$ we require that, for all $c \in CHAN$, $c! \in dch(S_1)$ implies $c? \notin dch(S_2)$, and $c? \in dch(S_1)$ implies $c! \notin dch(S_2)$.
- For a guarded command $G \equiv [\Box_{i=1}^{n} c_i? \to S_i \Box \mathbf{delay}\ d \to S_0]$ we require
  —for all $i \in \{1, \ldots, n\}$ that $c_i! \notin dch(G)$, and
  —for all $i, j \in \{0, 1, \ldots, n\}$, $i \neq j$, and $c \in CHAN$ that $c! \in dch(S_i)$ implies $c? \notin dch(S_j)$, and $c? \in dch(S_i)$ implies $c! \notin dch(S_j)$.
- For $S_1 \| S_2$ we require $dch(S_1) \cap dch(S_2) \subseteq CHAN$.

5.1c *Basic timing assumptions*: The precise meaning of this programming language is defined by a denotational semantics which describes the real-time behaviour of programs. Such real-time semantics requires information about implementation details from which one usually abstracts in non-real-time models, such as the execution time of assignments and the time required to evaluate boolean tests. Thus we have to make assumptions about the execution time of atomic statements and about the extra time needed to execute compound constructs, i.e., how the execution time of compound constructs can be obtained from the timing of the components. In our proof systems the correctness of a program with respect to a specification, which may include timing constraints, is verified relative to these assumptions.

In general we will have bounds on the execution time. Here we assume, for simplicity, that there is no overhead for compound statements and that a **delay** d statement takes exactly d time units. Furthermore we assume given a constant $K_c > 0$ such that each communication, i.e., without the waiting period, takes K_c time units.

The most important assumption involves parallel composition. To determine the execution time of parallel programs we need information about the progress of actions, representing the allocation of processes on processors. In general, we have to make an assumption about the execution model of parallel processes. In this paper we consider the *maximal parallelism* model to represent the situation that every process has its own processor. Hence, a process never waits with the execution of a local, non-communication, statement. An input or output statement can cause the process to wait, but only when no communication partner is available; as soon as both partners are available the communication must take place. Thus the maximal parallelism model implies a *minimal waiting* period.

In Hooman (1991a) we have generalized this maximal parallelism assumption to multiprogramming where several processes can be executed on a single processor and scheduling is based on priorities which can be assigned to statements in the program.

5.2 *Semantic model*

Our formal model of real-time communication behaviour consists of a mapping from points of time to sets of channel names, indicating the channels along which messages are being transmitted at any given time. In addition to the names of the channels along which a communication takes place, the model includes information about those processes waiting to send or waiting to receive messages on their incident channels at any given time. Using this information, the formalism enforces minimal waiting in our maximal parallelism model by requiring that no pair of processes is ever simultaneously waiting to send and waiting to receive, respectively, on a shared channel.

We express the timing behaviour of a program from the viewpoint of an external observer with his own clock (as done in Reed & Roscoe 1986, Koymans *et al* 1988). Thus, although parallel components of a system might have their own, physical, local clock, the observable behaviour of a system is described in terms of a single, conceptual, global clock. Since this global notion of time is not incorporated in the distributed system itself, it does not impose any synchronization upon processes. Then we define a real-time computation of a program by means of a function which assigns to a point of time a set of records, representing the events that are taking place at that point.

In this paper we use a time domain $TIME$ which is dense, i.e., between every two points of time there exists an intermediate point. With such a dense time domain a communication can be represented by an interval of time points during which we record this communication, and we can easily model communications that overlap in time or that are arbitrarily close to each other in time. Having dense time is also suitable for the description of reactive systems which interact with an environment that has a time-continuous nature (see, e.g. Koymans 1990). Furthermore, we argue that in a compositional framework it is inconvenient to use discrete time. Recall that compositionality allows us to design a process in isolation according to its specification. With a discrete notion of time a smallest time unit has to be chosen in this specification, and then when two independently developed processes with different time units are combined, a new basic time unit must be defined and the specifications of the processes have to be modified accordingly. Finally, a dense time domain allows the refinement of a single event into a sequence of sub-events, such as the implementation of a single synchronous communication by a sequence of asynchronous communications according to some protocol. An extensive discussion about the nature of time can be found in Joseph & Goswami (1989). In this paper we use the non-negative rationals as our (dense) time domain:

$$TIME = \{\tau \in \mathcal{Q} \mid \tau \geqslant 0\},$$

where $\mathcal{Q}$ is the set of rational numbers.

For notational convenience, a special value ∞ is used with the following properties: $\infty \notin TIME$, for all $\tau \in TIME$: $\tau < \infty$, and for all $\tau \in TIME \cup \{\infty\}$: $\tau + \infty = \infty + \tau = \infty$, $\infty - \tau = \infty$, $\tau \times \infty = \infty \times \tau = \infty$, $max(\infty, \tau) = max(\tau, \infty) = \infty$, $min(\infty, \tau) = min(\infty, \tau) = \tau$, and $min\ \phi = \infty$. For a point $\tau_0 \in TIME$, a left-closed right-open interval $[0, \tau_0)$ is defined as $\{\tau \mid \tau \in TIME \wedge 0 \leqslant \tau < \tau_0\}$.

DEFINITION 5.2 (Model)

Let $\tau_0 \in TIME \cup \{\infty\}$.
A *model* σ (of a real-time computation) is a mapping σ: $[0, \tau_0) \rightarrow \wp(DCHAN)$.

DEFINITION 5.3 (Length of a model)

For a model σ with domain $[0, \tau_0)$ the *length* of σ, denoted by $|\sigma|$, is defined as $|\sigma| = \tau_0$.

Thus, for all $\tau \in TIME$, with $\tau < |\sigma|$, we have $\sigma(\tau) \subseteq DCHAN$. Informally, a model σ represents the communication behaviour at each point in time during an execution of a program. If $|\sigma| = \infty$ then σ represents a non-terminating computation, and if $|\sigma| < \infty$ then it represents a computation that terminates at time $|\sigma|$. For a point of time τ, $\tau < |\sigma|$, and a channel name $c \in CHAN$, we have three possible elements $c, c!$ and $c?$ for $\sigma(\tau)$ with the following meaning:

- $c \in \sigma(\tau)$ if a communication takes place along channel c at time τ;
- $c! \in \sigma(\tau)$ if a process is waiting to send along channel c at time τ;
- $c? \in \sigma(\tau)$ if a process is waiting to receive along channel c at time τ.

DEFINITION 5.4 (Concatenation of models)

Define the *concatenation* of two models σ_1 and σ_2, denoted by $\sigma_1\sigma_2$, as

$$|\sigma_1\sigma_2| = |\sigma_1| + |\sigma_2|$$

and

$$\sigma_1\sigma_2(\tau) = \begin{cases} \sigma_1(\tau) & \text{for all } \tau < |\sigma_1| \\ \sigma_2(\tau - |\sigma_1|) & \text{for all } |\sigma_1| \leqslant \tau < |\sigma_1| + |\sigma_2| \end{cases}.$$

Note that, for all models σ_1 and σ_2, if $|\sigma_1| = \infty$ then $\sigma_1\sigma_2 = \sigma_1$.

A compositional semantics for our programming language is defined in Hooman (1991b) using the computational model as described above. The meaning of a program S, denoted by $\mathcal{M}(S)$, is a set of models representing the possible computations of S starting at time 0. $\mathcal{M}(S)$ is defined by induction on the structure of S according to the grammar in table 2.

5.3 *Specifications*

We modify the Hoare-style framework of the previous chapters by extending the first-order assertion language with primitives to specify the timing behaviour of programs. As already explained, by means of Hoare triples we can only specify partial correctness. Hence, we add a third assertion, called *commitment*, to specify real-time properties of terminating *and* non-terminating computations. In contrast with the previous chapters, the aim is to specify, besides safety properties (which can be falsified in infinite time), also liveness properties. Therefore the commitment will not be an invariant which holds at any point during a computation (as in §4), but it should hold for complete, possibly infinite, computations.

5.3a *Modification of Hoare triples to real-time*: To extend a Hoare triple $\{p\}S\{q\}$ to real-time, a special variable *time* is introduced. Consider, for instance the formula $\{time = 3\}$ **delay** $2\{time = 5\}$. In the pre-condition the variable *time* specifies the starting time of the program, whereas in the post-condition *time* denotes the termination time. Furthermore, to specify the timed communication behaviour of programs, the assertion language includes a primitive *comm via c at exp* to express that a communication along channel c takes place at time *exp*. As in the semantics, primitives are required to express that a process is waiting to communicate. Here we use *wait to c! at exp* to denote that a process is waiting to send a message along channel c at time *exp*, and *wait to c? at exp* to denote that a process is waiting to receive along channel c at *exp*. As usual in Hoare-style formalisms, logical variables are used to relate pre- and post-condition. It this chapter we have logical variables ranging over $TIME \cup \{\infty\}$, and quantification over these variables. For instance, with logical variable t, the specification $\{time = t\}S\{t + 4 < time < t + 7\}$ expresses that if S terminates then it takes between 4 and 7 time units.

Recall that a formula $\{p\}S\{q\}$ can only express the behaviour of terminating computations, and hence such a specification is trivially satisfied by non-terminating programs. Therefore we extend a Hoare triple $\{p\}S\{q\}$ with a third assertion, called a *commitment*, which expresses the real-time communication behaviour of all executions of S, including the non-terminating ones. This leads to a correctness formula of the form C: $\{p\}S\{q\}$. In general, commitment C reflects the real-time communication interface between parallel components, whereas the pre- and post-condition facilitate the reasoning for sequential composition and iteration.

Finally, we argue that termination should be expressible in commitments. Consider the statements $S_1 \equiv c?$ and $S_2 \equiv [c? \rightarrow \textbf{skip} \,\|\, c? \rightarrow \star[\textbf{delay } 1 \rightarrow \textbf{skip}]]$. Then the programs

S_1; *d*! and S_2; *d*!, which satisfy the same Hoare triples, can be distinguished in our extended framework by the commitment $\forall t_0$: (*comm via c at* $t_0 \rightarrow \exists t_1 \geqslant t_0$: [*wait to d*! *at* t_1 $\vee$ *comm via d at* t_1]). Since we aim at a compositional proof system, the distinction between S_1; *d*! and S_2; *d*! implies that S_1 and S_2 must also be distinguishable. Hence we have to express termination in the commitment. This can be done conveniently, without introducing new primitives, by allowing the special variable *time* to occur in commitments. Observe that the commitment can be seen as an extension of the post-condition to non-terminating computations. Hence, by interpreting *time* similar to post-conditions, *time* in commitments expresses the termination time of terminating computations. For non-terminating computations we use the special variable ∞ and such computations satisfy the commitment *time* $= \infty$. In the example above, S_1 and S_2 can be distinguished by using the commitment $\forall t_0$: (*comm via c at* $t_0 \rightarrow$ *time* $< \infty$).

5.3b *Assertion language*: Let *TVAR* be a set of logical variables ranging over $TIME \cup \{\infty\}$. Thy syntax of the assertion language is given in table 3, where $t \in TVAR$, $c \in CHAN$, and $\tau \in TIME \cup \{\infty\}$. Let *dch*(*p*) denote the set of all *c*, *c*!, or *c*? occurring in assertion *p*.

Table 3. Syntax of the assertion language.

Expression	$exp ::= \tau \mid t \mid time \mid exp_1 + exp_2 \mid exp_1 \times exp_2$
Assertion	$p ::=$ *comm via c at exp* $\mid$ *wait to c*! *at exp* $\mid$ *wait to c*? *at exp*
	$exp_1 = exp_2 \mid exp_1 < exp_2 \mid exp \in \mathcal{N} \mid$
	$\neg p \mid p_1 \vee p_2 \mid \exists t{:}p$

To interpret logical variables we use a logical variable environment $\gamma: TVAR \rightarrow TIME \cup \{\infty\}$, i.e., a mapping which assigns a value from $TIME \cup \{\infty\}$ to each logical variable. The value of a variable *t* in an environment γ is denoted by $\gamma(t)$. The *variant* of an environment γ with respect to a logical variable *t* and a value $\tau \in TIME \cup \{\infty\}$, denoted by $(\gamma: t \mapsto \tau)$, is defined as follows. For any logical variable t_1,

$$(\gamma: t \mapsto \tau)(t_1) = \begin{cases} \gamma(t_1) & \text{if } t \not\equiv t_1 \\ \tau & \text{if } t \equiv t_1 \end{cases}.$$

Then we formally define when an assertion *p* holds in an environment γ and a model σ as defined in § 5.2. The special variable *time* is interpreted as the length of σ(i.e., $|\sigma|$). If expression *exp* yields a value $\tau < |\sigma|$ then the interpretation of the primitives *comm via c at exp*, *wait to c*! *at exp*, and wait to *c*? at *exp* is straightforward using $\sigma(\tau)$. But if *exp* yields a value greater than $|\sigma|$ then we must be more careful with the meaning of these primitives. (Consider, for instance, *comm via c at* (*time* + 3). If such an assertion would hold in a model, which is intuitively strange, then this would lead to problems in the proof system. For instance, for a formula *C*:{*comm via c at*(*time* + 3)}*S*{*q*} the information from the pre-condition should not be used in *C* and *q*. In general, a pre-condition should express the behaviour before the start of a program and it should not restrict the behaviour of the program at points of time after the starting time. Thus we aim at an interpretation in which *comm via c at*(*time* + 3) never holds in a model. Note that *C*:{$\neg$ *comm via c at*(*time* + 3)}*S*{*q*} leads to the same problems, and hence also $\neg$ *comm via c at*(*time* + 3) should not hold in any model.

A possible solution is to be careful with negations and to apply it only to primitive assertions. Here we choose an alternative approach; to achieve a compositional definition of negation we use a three-valued interpretation. This means that the value of an assertion p in an environment γ and a model σ, denoted $[\![p]\!]\gamma\sigma$, is *true*, *false*, or $\perp$. To define negation and disjunction of assertions, logical operators NOT_3 and OR_3 for these three values are defined by the truth tables in table 4. These operators, which were introduced in Kleene (1952), are the strongest monotonic extensions of the classical (two-valued) operators.

Table 4. Three-valued negation and disjunction.

p	$NOT_3 p$
true	*false*
false	*true*
$\perp$	$\perp$

OR_3	*true*	*false*	$\perp$
true	*true*	*true*	*true*
false	*true*	*false*	$\perp$
$\perp$	*true*	$\perp$	$\perp$

First we define the value of expression *exp* in a model σ and an environment γ, denoted $v(exp)(\gamma,\sigma)$, yielding a value from $TIME\cup\{\infty\}$.

- $v(\tau)(\gamma,\sigma)=\tau$,
- $v(t)(\gamma,\sigma)=\gamma(t)$,
- $v(time)(\gamma,\sigma)=|\sigma|$,
- $v(exp_1+exp_2)(\gamma,\sigma)=v(exp_1)(\gamma,\sigma)+v(exp_2)(\gamma,\sigma)$,
- $v(exp_1\times exp_2)(\gamma,\sigma)=v(exp_1)(\gamma,\sigma)\times v(exp_2)(\gamma,\sigma)$.

Next we define inductively $[\![p]\!]\gamma\sigma$ as an element of $\{true, false, \perp\}$.

- $[\![comm\ via\ c\ at\ exp]\!]\gamma\sigma=\begin{cases} true, & \text{if } v(exp)(\gamma,\sigma)<|\sigma| \text{ and } c\in\sigma(exp)(\gamma,\sigma)) \\ false, & \text{if } v(exp)(\gamma,\sigma)<|\sigma| \text{ and } c\notin\sigma(v(exp)(\gamma,\sigma)), \\ \perp, & \text{if } v(exp)(\gamma,\sigma)\geqslant|\sigma| \end{cases}$

- $[\![wait\ to\ c!\ at\ exp]\!]\gamma\sigma=\begin{cases} true, & \text{if } v(exp)(\gamma,\sigma)<|\sigma| \text{ and } c!\in\sigma(v(exp)(\gamma,\sigma)) \\ false, & \text{if } v(exp)(\gamma,\sigma)<|\sigma| \text{ and } c!\notin\sigma(v(exp)(\gamma,\sigma)), \\ \perp, & \text{if } v(exp)(\gamma,\sigma)\geqslant|\sigma| \end{cases}$

- $[\![wait\ to\ c?\ at\ exp]\!]\gamma\sigma=\begin{cases} true, & \text{if } v(exp)(\gamma,\sigma)<|\sigma| \text{ and } c?\in\sigma(v(exp)(\gamma,\sigma)) \\ false, & \text{if } v(exp)(\gamma,\sigma)<|\sigma| \text{ and } c?\notin\sigma(v(exp)(\gamma,\sigma)), \\ \perp, & \text{if } v(exp)(\gamma,\sigma)\geqslant|\sigma| \end{cases}$

- $[\![exp_1=exp_2]\!]\gamma\sigma=\begin{cases} true, \text{ if } v(exp_1)(\gamma,\sigma)=v(exp_2)(\gamma,\sigma) \\ false, \text{ if } v(exp_1)(\gamma,\sigma)\neq v(exp_2)(\gamma,\sigma), \end{cases}$

- $[\![exp_1<exp_2]\!]\gamma\sigma=\begin{cases} true, \text{ if } v(exp_1)(\gamma,\sigma)<v(exp_2)(\gamma,\sigma) \\ false, \text{ if } v(exp_1)(\gamma,\sigma)\geqslant v(exp_1)(\gamma,\sigma), \end{cases}$

- $[\![exp\in\mathbb{N}]\!]\gamma\sigma=\begin{cases} true, \text{ if } v(exp)(\gamma,\sigma)\in\mathbb{N} \\ false, \text{ if } v(exp)(\gamma,\sigma)\notin\mathbb{N}' \end{cases}$

- $[\![\neg p]\!]\gamma\sigma=NOT_3[\![p]\!]\gamma\sigma$,
- $[\![p_1\vee p_2]\!]\gamma\sigma=[\![p_1]\!]\gamma\sigma OR_3[\![p_2]\!]\gamma\sigma$,

- $[\![\exists t{:}p]\!]\gamma\sigma=\begin{cases} true, & \text{if there exists a } \tau\in TIME\cup\{\infty\} \text{ with } [\![p]\!](\gamma{:}t\mapsto\tau)\sigma=true \\ false, & \text{if for all } \tau\in TIME\cup\{\infty\},\ [\![p]\!](\gamma{:}t\mapsto\tau)\sigma=false \\ \perp, & \text{otherwise.} \end{cases}$

The conventional abbreviations are used, such as $p_1 \wedge p_2 \equiv \neg(\neg p_1 \vee \neg p_2)$, $p_1 \rightarrow p_2 \equiv \neg p_1 \vee p_2$, and $\forall t{:}p \equiv \neg \exists t{:}\neg p$. Subsequently we say that *p holds in γ and σ* if $[\![p]\!]\gamma\sigma = true$. Furthermore, we frequently use $[\![p]\!]\gamma\sigma$ as an abbreviation of $[\![p]\!]\gamma\sigma = true$.

Returning to our example, observe that the interpretation is such that, for any γ and σ, $[\![comm\ via\ c\ at\ (time+3)]\!]\gamma\sigma = \bot$ and $[\![\neg\, comm\ via\ c\ at\ (time+3)]\!]\gamma\sigma = \bot$. Thus neither *comm via c at* $(time+3)$ nor $\neg$ *comm via c at* $(time+3)$ holds in γ and σ. In general this interpretation facilitates sequential reasoning, since $[\![p]\!]\gamma\sigma$ implies that p does not express any constraint on points of time after $|\sigma|$. This is expressed formally in lemma 5.5 below: if assertion p holds in σ_1 then p, with *time* replaced by $|\sigma_1|$, holds in any arbitrary extension of σ_1, i.e., in $\sigma_1\sigma_2$ for any σ_2. (A proof for this lemma can be found in Hooman 1991b.)

Lemma 5.5. For all γ *and* σ_1: $[\![p]\!]\gamma\sigma_1$ *iff (for all* σ_2, $[\![p[|\sigma_1|/time]]\!]\gamma\sigma_1\sigma_2$).

We use the conventional relations between expressions, such as

- $exp_1 \leqslant exp_2 \equiv (exp_1 < exp_2) \vee (exp_1 = exp_2)$,
- $exp_1 \geqslant exp_2 \equiv (exp_2 < exp_1) \vee (exp_1 = exp_2)$,
- $exp_1 \leqslant exp_2 \leqslant exp_3 \equiv (exp_1 \leqslant exp_2) \wedge (exp_2 \leqslant exp_3)$, etc.

Relativized quantifiers are defined as usual, for instance,

- $\forall t,\ t_0 \leqslant t < time{:}p \equiv \forall t{:}t_0 \leqslant t < time \rightarrow p$.
- $\exists t, t_0 \leqslant t < time{:}p \equiv \exists t{:}t_0 \leqslant t < time \wedge p$.

Furthermore, the following abbreviations are frequently used:

- $true \equiv 0 = 0$,
- $false \equiv \neg\, true$,
- *wait to c! during* $[t_0, t_1) \equiv \forall t_2,\ t_0 \leqslant t_2 < t_1$: *wait to c! at* t_2,
- *comm via c during* $[t_0, t_1) \equiv \forall t_2,\ t_0 \leqslant t_2 < t_1$:*comm via c at* t_2,
- *no comm via c during* $[t_0, t_1) \equiv \forall t_2,\ t_0 \leqslant t_2 < t_1{:}\neg$ *comm via c at* t_2,
- *wait to c! at* t_0 *until comm at* $t_1 \equiv$
  wait to c! during $[t_0, t_1) \wedge$ *comm via c during* $[t_1, t_1 + K_c)$,
- *wait to c! at* t_0 *until comm* $\equiv \exists t_1 \geqslant t_0$: *wait to c! at* t_0 *until comm at* t_1.

Let *cset* be a finite subset of *DCHAN*. Then

- *no cset during* $[t_0, t_1) \equiv \forall t,\ t_0 \leqslant t < t_1$: $\bigwedge_{c!\in cset} \neg$ *wait to c! at t* $\wedge$
  $\bigwedge_{c?\in cset} \neg$ *wait to c? at t* $\wedge \bigwedge_{c\in cset} \neg$ *comm via c at t*.

The abbreviations above are also used with *c*? instead of *c*!, and with other intervals such as (t_0, t_1) and (t_0, ∞) instead of the interval $[t_0, t_1)$. It is easy to extend these definitions for general expressions instead of t_0 or t_1.

Observe that logical variables range over $TIME \cup \{\infty\}$, and thus

> *wait to c! at* t_0 *until comm*, i.e., $\exists t_1 \geqslant t_0$: *wait to c! at* t_0 *until comm at* t_1 is equivalent to [*wait to c! at* t_0 *until comm at* ∞] $\vee$ [$\exists t_1,\ t_0 \leqslant t_1 < \infty$:*wait to c! at* t_0 *until comm at* t_1].

Since *comm via c during* $[\infty, \infty + K_c) \leftrightarrow true$, this is equivalent to

$$[wait\ to\ c!\ during[t_0, \infty)] \vee$$
$$[\exists t_1, t_0 \leqslant t_1 < \infty{:}\ wait\ to\ c!\ during[t_0, t_1) \wedge comm\ via\ c\ during\ [t_1, t_1 + K_c)].$$

DEFINITION 5.6. (Validity assertions)

An assertion p is *valid*, denoted $\models p$, iff p holds in any environment γ and any model σ, i.e., $[\![p]\!]\gamma\sigma$ for all γ and σ.

Next we define when a correctness formula C: $\{p\}S\{q\}$ is valid. Informally, if p holds in an initial model $\hat{\sigma}$, and σ represents a computation of S, then C holds in the concatenation $\hat{\sigma}\sigma$ of these models, and if σ terminates then q holds in $\hat{\sigma}\sigma$. This leads to the following formal definition (recall that, e.g., $[\![p]\!]\gamma\hat{\sigma}$ is an abbreviation of $[\![p]\!]\gamma\hat{\sigma} = true$).

DEFINITION 5.7. (Validity of a correctness formula)

For a program S and assertions C, p and q, a correctness formula C:$\{p\}S\{q\}$ is *valid*, denoted by $\models C$:$\{p\}S\{q\}$, iff for any γ and any well-formed model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$:

$$\text{if } [\![p]\!]\gamma\hat{\sigma} \text{ then for all } \sigma \in \mathcal{M}(S): [\![C]\!]\gamma(\hat{\sigma}\sigma), \text{ and if } |\hat{\sigma}| < \infty \text{ then } [\![q]\!]\gamma(\hat{\sigma}\sigma).$$

Example 5.2. We show that $\models time = t + d$:$\{time = t\}$**delay** $d\{time = t + d\}$. Consider an environment γ and a model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$. Assume $[\![time = t]\!]\gamma\hat{\sigma}$. Then $|\hat{\sigma}| = \gamma(t)$, i.e., the starting time is the value of t in environment γ. For $\sigma \in \mathcal{M}$(**delay** d) we have $|\sigma| = d$. Then $[\![time = t + d]\!]\gamma\hat{\sigma}\sigma$, since $|\hat{\sigma}\sigma| = |\hat{\sigma}| + |\sigma| = \gamma(t) + d$. □

Observe that the definition of validity of a correctness formula requires that the assertions hold for each environment γ, and hence free logical variables in a specification are implicitly universally quantified.

Example 5.3. As an example of a liveness specification, consider the formula

$$C \wedge time = \infty\text{: } \{time = 0\} \star[c? \rightarrow \textbf{skip}]\{false\},$$

with

$$C \equiv (\forall t_0 < \infty \exists t_1, > t_0\text{: } comm\ via\ c\ at\ t_1] \vee$$
$$(\exists t_0 < \infty\text{: } wait\ to\ c?\ during[t_0, \infty)).$$

This commitment expresses that the program either communicates infinitely often, or it eventually waits forever. Observe that this is not a safety property since it cannot be falsified in finite time. After presenting the rule for the iteration construct we show that this valid formula is also derivable, as it should be in a complete proof system. □

5.4 *Proof system*

In this section we give a compositional proof system for our correctness formulae. First we formulate rules and axioms that are generally applicable to any statement. Next we axiomatize the programming language by formulating rules and axioms for all atomic statements and compounds programming constructs. Let $\vdash C$:$\{p\}S\{q\}$ denote that the formula C:$\{p\}S\{q\}$ can be derived in this proof system.

General part

We start with an axiom expressing the well-formedness properties of a computation. Let *cset* be a finite subset of *DCHAN*.

Axiom 5.8. (Well-formedness)

$$\mathit{Well\ Form}_{cset}: \{true\}\, S\, \{\mathit{Well\ Form}_{cset}\}$$

where

$$\mathit{Well\ Form}_{cset} \equiv \forall t < time: M\,W_{cset}(t) \wedge Excl_{cset}(t),$$

with

$$M\,W_{cset}(t) \equiv \bigwedge_{\{c!, c?\} \subseteq cset} \neg(\mathit{wait\ to\ c?\ at\ t} \wedge \mathit{wait\ to\ c!\ at\ t}),$$

$$Excl_{cset}(t) \equiv \left(\bigwedge_{\{c, c!\} \subseteq cset} \neg(\mathit{wait\ to\ c!\ at\ t} \wedge \mathit{comm\ via\ c\ at\ t}) \right) \wedge$$
$$\left(\bigwedge_{\{c, c?\} \subseteq cset} \neg(\mathit{wait\ to\ c?\ at\ t} \wedge \mathit{comm\ via\ c\ at\ t}) \right).$$

We give a few lemmas concerning the use of $\mathit{Well\ Form}_{cset}$. These lemmas will be used in the chapter 6 where we illustrate our formalism by an example of a watchdog timer.

Lemma 5.9. *For all* t_0, t_1,

$$\mathit{wait\ to\ c?\ during}(t_0, t_1) \wedge \mathit{Well\ Form}_{\{c, c!, c?\}} \rightarrow no\{c!, c\}\ \mathit{during}\ (t_0, t_1).$$

Lemma 5.10. *For all* t_0,

$$\mathit{wait\ to\ c!\ at}\ t_0\ \mathit{until\ comm} \wedge \mathit{Well\ Form}_{\{c, c!, c?\}} \rightarrow$$
$$\forall t_1,\ t_0 < t_1 < t_0 + K_c: \neg\, \mathit{wait\ to\ c?\ at}\ t_1.$$

The proof system contains a consequence rule which is an extension of the classical consequence rule for Hoare triples (see rule 3.3 in § 3). Note that by definition 5.7 of a valid correctness formula, pre-conditions are interpreted in a model $\hat{\sigma}$ with $|\hat{\sigma}| < \infty$. Hence any pre-condition can be strengthened by adding $time < \infty$ to express that the starting time if finite.

Rule 5.11. (Consequence)

$$\frac{C_0: \{p_0\}\, S\, \{q_0\},\ p \wedge time < \infty \rightarrow p_0,\ C_0 \rightarrow C,\ q_0 \rightarrow q}{C: \{p\}\, S\, \{q\}}.$$

Observe that $\models$ *false*: $\{time = \infty\}\, S\, \{false\}$, for any program S. To deduce this formula, we first derive *false*: $\{false\}\, S\, \{false\}$. This can be done by means of the initial invariance rule below. Then we can use the consequence rule, since $time = \infty \wedge time < \infty \rightarrow false$.

Next we give two axioms to deduce invariance properties. The first axiom expresses that the pre-condition, except for the variable *time*, remains valid during the execution of a program.

Axiom 5.12. (Initial invariance)

$$p: \{p\}\, S\, \{p\}$$

provided *time* does not occur in p.

The soundness of this axiom is based on lemma 5.5 which guarantees that if $[\![p]\!]\gamma\hat{\sigma}$ and p does not contain *time*, then $[\![p]\!]\gamma\hat{\sigma}\sigma$, for any model σ.

The channel invariance axiom below expresses that during the execution of a program S no activity takes place on channels not occurring in S. Let *cset* be a finite subset of *DCHAN*.

Axiom 5.13. (Channel invariance)

$$no\ cset\ during\ [t_0, time){:}\{time = t_0\}S\{no\ cset\ during[t_0, time)\}$$

provided $cset \cap dch(S) = \emptyset$.

Our proof system contains the following rules for conjunction, quantification, and substitution.

Rule 5.14. (Conjunction)

$$\frac{C_1{:}\{p_1\}S\{q_1\},\ C_2{:}\{p_2\}S\{q_2\}}{C_1 \wedge C_2{:}\{p_1 \wedge p_2\}S\{q_1 \wedge q_2\}}.$$

Rule 5.15. (Quantification)

$$\frac{C{:}\{p\}S\{q\}}{C{:}\{\exists t{:}p\}S\{q\}}$$

provided t does not occur in C and q.

Rule 5.16. (Substitution)

$$\frac{C{:}\{p\}S\{q\}}{C[exp/t]{:}\{p[exp/t]\}S\{q[exp/t]\}}$$

provided *time* does not occur in expression *exp*.

The following rule can be used to transform a correctness formula with pre-condition $time = t_0$ into a formula with an arbitrary pre-condition and starting time. This is a derived rule, that is, the rule can derived from the other rules and axioms in the proof system (as we prove below).

Derived rule 5.17. (Adaptation)

$$\frac{C{:}\{time = t_0\}S\{q\}}{\begin{array}{r}p[exp/time] \wedge C[exp/t_0]{:}\{p[exp/time] \wedge time = exp\}\\ S\{p[exp/time] \wedge q[exp/t_0]\}\end{array}}$$

provided *time* does not occur in expression *exp*.

Proof. Assume $\vdash C{:}\{time = t_0\}S\{q\}$, and suppose *time* does not occur in expression *exp*. By the substitution rule, replacing t_0 by *exp*, we obtain

$$\vdash C[exp/t_0]{:}\{time = exp\}S\{q[exp/t_0]\}$$

Since *time* does not occur in *exp*, the initial invariance rule leads to

$$\vdash p[exp/time]\colon\{p[exp/time]\}S\{p[exp/time]\}$$

Then, by the consequence rule,

$$\vdash p[exp/time] \wedge C[exp/t_0]\colon\{p[exp/time] \wedge time = exp\}$$
$$S\{p[exp/time] \wedge q[exp/t_0]\} \qquad \square$$

Program part

The rules and axioms for atomic statements will be given with pre-condition $time = t_0$; with the adaptation rule one can easily obtain any arbitrary pre-condition.

Axiom 5.18. (Skip)

$$time = t_0\colon\{time = t_0\}\mathbf{skip}\{time = t_0\}$$

Axiom 5.19. (Delay)

$$time = t_0 + d\colon\{time = t_0\}\ \mathbf{delay}\ d\{time = t_0 + d\}$$

Example 5.4. We can derive

$$(time = 5 \wedge comm\ via\ c\ at\ 1)\colon\{time = 2 \wedge comm\ via\ c\ at\ 1\}\mathbf{delay}\ 3\{time = 5\}$$

as follows. By the delay axiom,

$$time = t_0 + 3\colon\{time = t_0\}\mathbf{delay}\ 3\{time = t_0 + 3\}.$$

Using the adaptation rule, with $p \equiv comm\ via\ c\ at\ 1$ and $exp = 2$, we obtain

$$(time = 2 + 3 \wedge comm\ via\ c\ at\ 1)\colon$$
$$\{time = 2 \wedge comm\ via\ c\ at\ 1\}\mathbf{delay}\ 3\{time = 2 + 3 \wedge comm\ via\ c\ at\ 1\}.$$

Finally, the consequence rule leads to

$$(time = 5 \wedge comm\ via\ c\ at\ 1)\colon\{time = 2 \wedge comm\ via\ c\ at\ 1\}\mathbf{delay}\ 3\{time = 5\}.$$
$\square$

To formulate a rule for a send statement *c*!, observe that the post-condition can characterize terminating computations consisting of a waiting period (during which no communication partner is available) followed by an interval during which the actual communication takes place. In addition, the commitment can characterize non-terminating computations in which the i/o-statement waits forever to communicate. Observe that $\exists t \geqslant t_0$: *wait to c! at* t_0 *until comm at t* $\wedge$ $time = t + K_c$ implies that either $t = \infty$ and *wait to c! during* $[t_0, \infty) \wedge time = \infty$, or there exists a $t, t_0 \leqslant t < \infty$ such that *wait to c! during*$[t_0, t) \wedge$ *comm via c during*$[t, t + K_c) \wedge time = t + K_c$. This leads to the following rule:

Rule 5.20. (Send)

$$\frac{\exists t \geqslant t_0 : \textit{wait to } c! \textit{ at } t_0 \textit{ until comm at } t \wedge \textit{time} = t + K_c \rightarrow C}{C : \{\textit{time} = t_0\} c! \{C \wedge \textit{time} < \infty\}}$$

Similar to the send rule, we have the following rule for a receive statement.

Rule 5.21. (Receive)

$$\frac{\exists t \geqslant t_0 : \textit{wait to } c? \textit{ at } t_0 \textit{ until comm at } t \wedge \textit{time} = t + K_c \rightarrow C}{C : \{\textit{time} = t_0\} c? \{C \wedge \textit{time} < \infty\}}.$$

The inference rule for sequential composition is an extension of the classical rule for Hoare triples. To explain the commitment of S_1; S_2, observe that a computation of S_1; S_2 is either a non-terminating computation of S_1 or a terminated computation of S_1 extended with a computation of S_2. The commitment of S_1; S_2 expresses the non-terminating computations of S_1 by using the commitment of S_1 with $\textit{time} = \infty$. Terminating computations of S_1 are characterized in the post-conditions of S_1 which is also the pre-condition of S_2. Then these computations are extended by S_2 and described in the commitment of S_2.

Rule 5.22. (Sequential composition)

$$\frac{C_1 : \{p\} S_1 \{r\},\ C_2 : \{r\} S_2 \{q\}}{(C_1 \wedge \textit{time} = \infty) \vee C_2 : \{p\} S_1 ; S_2 \{q\}}.$$

Example 5.5. Consider the program $c?; d!$. Define

$$C^1_{nonterm} \equiv \textit{wait to } c? \textit{ during } [0, \infty)_1 \text{ and } C^1_{term} \equiv \textit{wait to } c? \textit{ at } 0 \textit{ until comm}_2 \textit{ at } t_1.$$

Then

$$(C^1_{nonterm} \wedge \textit{time} = \infty) \vee (\exists t_1 < \infty : C^1_{term} \wedge \textit{time} = t_1 + K_c) : \{\textit{time} = 0\} c? \{\exists t_1 < \infty : C^1_{term} \wedge \textit{time} = t_1 + K_c\}.$$

For $d!$, define $C_2 \equiv \textit{wait to } d! \textit{ at } t_1 + K_c \textit{ until comm}$. Then we can derive

$$(\exists t_1 < \infty : C^1_{term} \wedge C_2) : \{\exists t_1 < \infty : C^1_{term} \wedge \textit{time} = t_1 + K_c\} d! \{\textit{true}\}.$$

Observe that the terminating behaviour of $c?$ is characterized by its post-condition, thus by the pre-condition of $d!$, and hence can be included in the commitment of $d!$. Now the sequential composition rule leads to

$$(C^1_{nonterm} \wedge \textit{time} = \infty) \vee (\exists t_1 < \infty : C^1_{term} \wedge C_2) : \{\textit{time} = 0\} c?; d! \{\textit{true}\}. \qquad \square$$

Given the rules for the basic statements, it is often easier to use the following derived rule:

Derived rule 5.23. (Sequential composition adaptation)

$$\frac{C_1 : \{p\} S_1 \{r\},\ C_2 : \{\textit{time} = t\} S_2 \{q\}}{(C_1 \wedge \textit{time} = \infty) \vee (\exists t : r[t/\textit{time}] \wedge C_2) : \{p\} S_1 ; S_2 \{\exists t : r[t/\textit{time}] \wedge q\}}$$

Proof. Assume

$$\vdash C_1{:}\{p\}S_1\{r\}, \tag{1}$$

$$\vdash C_2{:}\{time = t\}S_2\{q\}. \tag{2}$$

Since $r \to \exists t{:}r[t/time] \wedge time = t$, (1) leads by the consequence rule to

$$\vdash C_1{:}\{p\}S_1\{\exists t{:}r[t/time] \wedge time = t\}. \tag{3}$$

By (2) and the adaptation rule:

$$\vdash r[t/time] \wedge C_2{:}\{r[t/time] \wedge time = t\}S_2\{r[t/time] \wedge q\}.$$

The consequence rule leads to

$$\vdash (\exists t{:}r[t/time] \wedge C_2){:}\{r[t/time] \wedge time = t\}S_2\{\exists t{:}r[t/time] \wedge q\}.$$

Then, using the quantification rule, we obtain

$$\vdash (\exists t{:}r[t/time] \wedge C_2){:}\{\exists t{:}r[t/time] \wedge time = t\}S_2\{\exists t{:}r[t/time] \wedge q\}. \tag{4}$$

From (3) and (4), by the sequential composition rule,

$$\vdash (C_1 \wedge time = \infty) \vee (\exists t{:}r[t/time] \wedge C_2){:}\{p\}S_1{;}S_2\{\exists t{:}r[t/time] \wedge q\}. \quad \Box$$

Example 5.6. Consider the program $c?;\ d!$. We prove

$(\exists t_1 \geqslant t_0{:}\textit{wait to c? at } t_0 \textit{ until comm at } t_1 \wedge \textit{wait to d! at } t_1 + K_c \textit{ until comm})$:
$\{time = t_0\}c?;\ d!\ \{true\}$

Let $C_1 \equiv \exists t_1 \geqslant t_0{:}\textit{wait to c? at } t_0 \textit{ until comm at } t_1 \wedge time = t_1 + K_c$, then by the receive rule we can derive

$$C_1{:}\{time = t_0\}c?\{C_1 \wedge time < \infty\}.$$

Let $C_2 \equiv \textit{wait to d! at t until comm}$, then from the send rule we obtain

$$C_2{:}\{time = t\}d!\{true\}.$$

By the derived sequential composition rule we can now derive

$$(C_1 \wedge time = \infty) \vee (\exists t{:}(C_1 \wedge time < \infty)[t/time] \wedge C_2){:}$$
$$\{time = t_0\}c?;\ d!\{\exists t{:}(C_1 \wedge time < \infty)[t/time] \wedge true\}.$$

Observe that the commitment $(C_1 \wedge time = \infty) \vee (\exists t{:}(C_1 \wedge time < \infty)[t/time] \wedge C_2)$ implies

$[\textit{wait to c? at } t_0 \textit{ until comm at } \infty \wedge time = \infty] \vee$
$[\exists t \exists t_1 \geqslant t_0{:}\textit{wait to c? at } t_0 \textit{ until comm at } t_1 \wedge t = t_1 + K_c \wedge t < \infty \wedge$
$\textit{wait to d! at t until comm}]$,

and thus

$[\textit{wait to c? at } t_0 \textit{ until comm at } \infty \wedge time = \infty] \vee$
$[\exists t_1, t_0 \leqslant t_1 < \infty{:}\ \textit{wait to c? at } t_0 \textit{ until comm at } t_1 \wedge \textit{wait to d! at } t_1 + K_c \textit{ until comm}]$.

Since $\models$ *wait to d! at* ∞ *until comm*, the consequence rule leads to
$(\exists t_1 \geqslant t_0{:}\textit{wait to } c? \textit{ at } t_0 \textit{ until comm at } t_1 \wedge \textit{wait to } d! \textit{ at } t_1 + K_c \textit{ until comm})$:

$$\{time = t_0\} c?;\ d! \{true\} \qquad \square$$

Consider a guarded command $G \equiv [\square_{i=1}^{n} c_i? \rightarrow S_i \square \mathbf{delay}\ d \rightarrow S]$. Define

- $\textit{wait in } G \textit{ during}[t_0, t) \equiv \bigwedge_{i=1}^{n} \textit{wait to } c_i? \textit{ during } [t_0, t) \wedge$
$no(dch(G) - \{c_1?, \ldots, c_n?\}) \textit{ during}[t_0, t)$
- $\textit{comm } c_i \textit{ in } G \textit{ from } t \equiv \textit{comm via } c_i \textit{ during } [t, t + K_c) \wedge$
$no(dch(G) - \{c_i\}) \textit{during}[t, t + K_c) \wedge time = t + K_c$

First we give a rule for the case that $d = \infty$, thus for $G \equiv [\square_{i=1}^{n} c_i? \rightarrow S_i]$. This statement either waits forever to perform one of the $c_i?$ communications because none of the partners is available, or it eventually communicates via one of the $c_i?$ and then executes the corresponding statement S_i.

Rule 5.24. (Guarded command without delay)

$$\frac{\begin{array}{l} \textit{wait in } G \textit{ during } [t_0, \infty) \wedge time = \infty \rightarrow C_{nonterm} \\ \exists t,\ t_0 \leqslant t < \infty{:}\ \textit{wait in } G \textit{ during } [t_0, t) \wedge \textit{comm } c_i \textit{ in } G \textit{ from } t \rightarrow p_i, \\ \text{for all } i = 1, \ldots, n \\ C_i{:}\{p_i\} S_i \{q_i\}, \text{ for all } i = 1, \ldots, n \end{array}}{C_{nonterm} \vee \bigvee_{i=1}^{n} C_i{:}\{time = t_0\} [\square_{i=1}^{n} c_i? \rightarrow S_i] \{\bigvee_{i=1}^{n} q_i}$$

Next consider $G \equiv [\square_{i=1}^{n} c_i? \rightarrow S_i \square \mathbf{delay}\ d \rightarrow S]$ with $d \neq \infty$.

Rule 5.25. (Guarded command with delay)

$$\frac{\begin{array}{l} \exists t, t_0 \leqslant t < t_0 + d{:}\textit{wait in } G \textit{ during } [t_0, t] \wedge \\ \qquad \textit{comm } c_i \textit{ in } G \textit{ from } t \rightarrow p_i, \text{ for all } i = 1, \ldots, n \\ C_i{:}\{p_i\} S_i \{q_i\}, \text{ for all } i = 1, \ldots, n \\ C{:}\{\textit{wait in } G \textit{ during}[t_0,\ t_0 + d) \wedge time = t_0 + d\} S \{q\} \end{array}}{\bigvee_{i=1}^{n} C_i \vee C{:}\{time = t_0\} [\square_{i=1}^{n} c_i? \rightarrow S_i \square \mathbf{delay}\ d \rightarrow S] \{\bigvee_{i=1}^{n} q_i \vee q\}}$$

provided $d \neq \infty$.

The rule for the iteration construct does not contain any explicit well-foundedness argument, although we deal with liveness properties. The main principle is that liveness properties can be derived from real-time safety properties, and these properties can be proved by means of an invariant (see example 5.7),

Rule 5.26. (Iteration)

$$\frac{\begin{array}{l} C{:}\{C\} G \{C\}. \\ (\forall t_1 < \infty \exists t_2 > t_1{:}C[t_2/time]) \rightarrow C_{nonterm} \end{array}}{C_{nonterm} \wedge time = \infty{:}\{C\} {\star} G \{false\}}$$

where t_1 and t_2 are fresh logical variables.

Example 5.7. Consider the formula from example 5.3, expressing a liveness property for program $\star[c? \rightarrow \mathbf{skip}]$. Let $C_1 \equiv \forall t_3 < \infty \exists t_4 > t_3{:}\mathit{comm\ via\ c\ at\ t_4}$ and

$$C_2 \equiv \exists t_3 < \infty{:}\mathit{wait\ to\ c?\ during}[t_3, \infty).$$

To prove $(C_1 \vee C_2) \wedge \mathit{time} = \infty{:}\{\mathit{time} = 0\}\ \star[c? \rightarrow \mathbf{skip}]\{\mathit{false}\}$, we apply the iteration rule with $C_{\mathit{nonterm}} \equiv C_1 \vee C_2$ and $C \equiv (\forall t_3 < \mathit{time} \exists t_4 > t_3{:}\ \mathit{comm\ via\ c\ at\ t_4}) \vee (\exists t_3 < \mathit{time}{:}\ \mathit{wait\ to\ c?\ during}\ [t_3, \infty))$.

Observe that C expresses that $C_1 \vee C_2$ holds up to the termination time. We show that the two conditions of the iteration rule are fulfilled:

1. We prove $C{:}\{C\}[c? \rightarrow \mathbf{skip}]\{C\}$ as follows.
By the rule for guarded command without delay we obtain

$$\hat{C}{:}\{\mathit{time} = t_0\}[c? \rightarrow \mathbf{skip}]\{\hat{C}\},$$

with

$$\hat{C} \equiv (\forall t_5,\ t_0 \leqslant t_5 < \mathit{time} \exists t_6 > t_5{:}\mathit{comm\ via\ c\ at\ t_6}) \vee$$
$$(\exists t_5,\ t_0 \leqslant t_5 < \mathit{time}{:}\ \mathit{wait\ to\ c?\ during}[t_5, \infty)).$$

From the adaptation rule, with $p \equiv C$ and $\mathit{exp} \equiv t_0$,

$$C[t_0/\mathit{time}] \wedge \hat{C}{:}\{C[t_0/\mathit{time}] \wedge \mathit{time} = t_0\}[c? \rightarrow \mathbf{skip}]\{C[t_0/\mathit{time}] \wedge \hat{C}\}.$$

Since $C[t_0/\mathit{time}] \wedge \hat{C} \rightarrow C$, the consequence rule leads to

$$C{:}\{C[t_0/\mathit{time}] \wedge \mathit{time} = t_0\}[c? \rightarrow \mathbf{skip}]\{C\}.$$

By the quantification rule,

$$C{:}\{\exists t_0{:}C[t_0/\mathit{time}] \wedge \mathit{time} = t_0\}[c? \rightarrow \mathbf{skip}]\{C\}.$$

Since $C \rightarrow \exists t_0{:}C[t_0/\mathit{time}] \wedge \mathit{time} = t_0$, the consequence rule leads to

$$C{:}\{C\}[c? \rightarrow \mathbf{skip}]\{C\}.$$

2. We have $(\forall t_1 < \infty \exists t_2 > t_1{:}C[t_2/\mathit{time}]) \rightarrow C_{\mathit{nonterm}}$, since

$$(\forall t_1 < \infty \exists t_2 > t_1{:}C[t_2/\mathit{time}]) \equiv (\forall t_1 < \infty \exists t_2 > t_1{:}$$
$$[\forall t_3 < t_2 \exists t_4 > t_3{:}\mathit{comm\ via\ c\ at\ t_4}] \vee [\exists t_3 < t_2{:}\ \mathit{wait\ to\ c?\ during}[t_3, \infty)]) \rightarrow$$
$$([\forall t_3 < \infty \exists t_4 > t_3{:}\mathit{comm\ via\ c\ at\ t_4}] \vee [\exists t_3 < \infty{:}\mathit{wait\ to\ c?\ during}[t_3, \infty)]) \equiv$$
$$(C_1 \vee C_2) \equiv C_{\mathit{nonterm}}.$$

Now by the iteration rule we obtain $(C_1 \vee C_2) \wedge \mathit{time} = \infty{:}\{C\}\star[c? \rightarrow \mathbf{skip}]\{\mathit{false}\}$. Since logical time-variables such as t_3 and t_4 range over nonnegative values, $\mathit{time} = 0$ implies $\forall t_3 < \mathit{time} \exists t_4 > t_3{:}\ \mathit{comm\ via\ c\ at\ t_4}$, and hence $\mathit{time} = 0 \rightarrow C$. Thus, by the consequence rule, $(C_1 \vee C_2) \wedge \mathit{time} = \infty{:}\{\mathit{time} = 0\}\star[c? \rightarrow \mathbf{skip}]\{\mathit{false}\}$. □

Consider the parallel composition of statements S_1 and S_2. For the pre-conditions we simply take the conjunction. For the post-condition q of $S_1 \| S_2$ we would also prefer to take the conjunction of the post-conditions q_1 and q_2 of, respectively S_1 and S_2, but there a small problem has to be solved. Observe that, for $i = 1, 2$, the special variable *time* in post-condition q_i of S_i denotes the termination time of S_i. Since, in general, the termination times of S_1 and S_2 will be different (and then $q_1 \wedge q_2$ could imply *false*, see example 5.8), we substitute a logical variable t_i for *time* in q_i. Then the termination time of $S_1 \| S_2$, expressed by *time* in its post-condition, is the maximum of t_1 and t_2. Furthermore we add two predicates to express that process S_i does not perform any action between t_i and *time*. A similar construction is used for the commitments. This leads to the following rule:

Rule 5.27. (Parallel composition)

$$\frac{\begin{array}{l} C_i\!:\{p_i\}S_i\{q_i\},\ i=1,2 \\ \exists t_1,t_2\!:time=max(t_1,t_2) \wedge \bigwedge_{i=1}^{2} C_i[t_i/time] \wedge no\ dch(S_i) during[t_i, time) \rightarrow C \\ \exists t_1,t_2\!:time=max(t_1,t_2) \wedge \bigwedge_{i=1}^{2} q_i[t_i/time] \wedge no\ dch(S_i) during[t_i, time) \rightarrow q \end{array}}{C\!:\{p_1 \wedge p_2\}S_1 \| S_2\{q\}}$$

provided t_1 and t_2 are fresh logical variables, and $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1,2\}$.

Example 5.8. To illustrate the problem with the termination times at parallel composition, consider the following two (valid) formulae:

$$time=5\!:\{time=0\}\,\mathbf{delay}\ 5\{time=5\},$$

and

$$time=7\!:\{time=0\}\,\mathbf{delay}\ 7\{time=7\}.$$

Then for **delay** 5 ‖ **delay** 7 we cannot take the conjunction of commitments and post-conditions, but by the rule above we obtain the commitment and post-condition $time=7$ because $(\exists t_1, t_2\!:time=max(t_1,t_2) \wedge t_1=5 \wedge t_2=7) \rightarrow (time=7)$. □

If *time* does not occur in commitments and post-conditions of the components S_1 and S_2 then we can derive from rule 5.27 the following simple rule:

Derived rule 5.28. (Simple parallel composition)

$$\frac{C_1\!:\{p_1\}S_1\{q_1\},\ C_2\!:\{p_2\}S_2\{q_2\}}{C_1 \wedge C_2\!:\{p_1 \wedge p_2\}S_1 \| S_2\{q_1 \wedge q_2\}}$$

provided $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1,2\}$, and *time* does not occur in C_1, C_2, q_1, and q_2.

6. Example – watchdog timer

The formalism from § 5 is illustrated by an example of a watchdog timer. Consider the network pictured in figure 1. Process W is a "watchdog" process: its job is to ensure that processes $P_1, \ldots, P_n$ are functioning properly. We abstract from the task that has to be performed by P_i, but we assume that P_i is functioning correctly iff it is ready to send (or sending) a reset signal on channel re_i to W at least once every v_i time units, for some constant v_i. So long as all processes P_i are ready to send a reset signal in time, watchdog timer W communicates on each re_i at least once every v_i time units and then it does not communicate on channel *al*. As soon as W has to

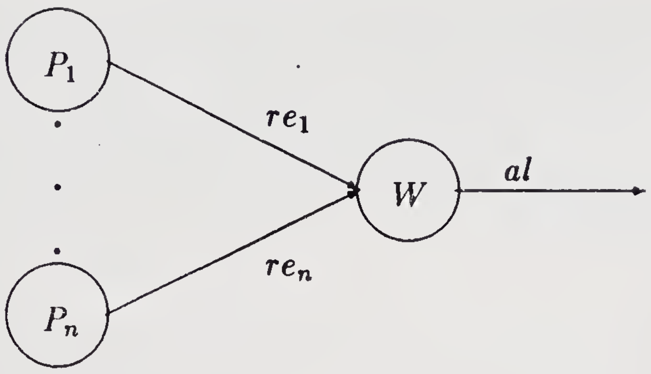


Figure 1. Watchdog timer network.

wait for a reset signal on a particular re_k during v_k time units, then it is ready to send (or sending) an alarm message on channel *al* within, say, K time units.

In this section we first give a formal specification for process W. Then, given specifications for the P_i, we prove that $P_1 \| \ldots \| P_n \| W$ is ready to send (or sending) on channel *al* iff one of the P_i is not functioning correctly. This is verified using our proof system without knowing the implementations of $P_1, \ldots P_n$ and W. To demonstrate program design from a specification, W is implemented as a parallel composition $W_1 \| \ldots \| W_n \| A$, and we derive the specification of W using specifications for W_i and A. Next W_i and A are, independently, implemented, and we prove that these programs satisfy the corresponding specifications.

6.1 *Specification of the watchdog timer*

We give a formal specification for the watchdog timer W and derive properties from it, using certain specifications for the processes P_i. In the specification of W we express that if there is a waiting period of v_k time units to receive input via re_k then, for some constant K, W starts waiting to send on channel *al* within K until the actual communication takes place. Furthermore, W tries to communicate via channel *al* at a certain point of time only if, for some k, there was a previous period of at least v_k time units during which W is waiting to receive input via re_k. Let

$$C_0^W \equiv \forall t_0 < \infty\colon \textit{wait to } re_k? \textit{ during } (t_0, t_0 + v_k) \rightarrow$$
$$(\exists t \leqslant t_0 + v_k + K\colon \textit{wait to al! at } t \textit{ until comm}),$$
$$C_1^W \equiv \forall t_1 < \infty\colon \textit{wait to al! at } t_1 \textit{until comm} \rightarrow$$
$$(\exists k \exists t_2 \leqslant t_1\colon \textit{wait to } re_k? \textit{ during } (t_2, t_2 + v_k)).$$

Then we specify W by $C_0^W \wedge C_1^W\colon \{\textit{time} = 0\}\, W \{\textit{true}\}$.

We prove that W tries to send a message via *al* iff there is an error in one of the processes P_i. Therefore we assume given a specification for P_i in which we use a predicate $\textit{error}_i$ representing some erroneous behaviour of P_i. Thus assume that, for all i, we have $C^{P_i}\colon \{\textit{time} = 0\}\, P_i \{\textit{true}\}$, where

$$C^{P_i} \equiv \textit{error}_i \leftrightarrow (\exists t_0 < \infty\colon \textit{no}\, \{re_i!, re_i\}\, \textit{during } (t_0, t_0 + v_i)).$$

This asserts that there is an error in P_i iff there exists a period of v_i time units during which P_i is not communicating via re_i and not waiting to communicate via re_i. Given our specifications for $P_1, \ldots, P_n$ and W, we try to prove that $P_1 \| \ldots \| P_n \| W$ satisfies the commitment $(\exists k{:}error_k) \leftrightarrow (\exists t < \infty{:}wait\ to\ al!\ at\ t\ until\ comm)$. Applying the simple parallel composition rule n times we obtain

$$C_0^W \wedge C_1^W \wedge \bigwedge_{i=1}^n C^{P_i}{:}\{time = 0\}\, P_1 \| \ldots \| P_n \| W \{true\}.$$

By the well-formedness axiom and the consequence rule we can derive

$$Well\ Form_{\{re_k, re_k!, re_k? | k = 1,\ldots,n\}}{:}\{true\}\, P_1 \| \ldots \| P_n \| W(true\}.$$

Using the conjunction rule we obtain the following commitment:

$$C_0^W \wedge C_1^W \wedge \bigwedge_{i=1}^n C^{P_i} \wedge Well\ Form_{\{re_k, re_k!, re_k? | k = 1,\ldots,n\}}.$$

First we prove that this commitment implies

$$(\exists t < \infty{:}wait\ to\ al!\ at\ t\ until\ comm) \rightarrow (\exists k{:}error_k).$$

$$\begin{array}{ll}
 & \exists t < \infty{:}wait\ to\ al!\ at\ t\ until\ comm \\
\Rightarrow \{C_1^W\} & \exists t < \infty\, \exists k \exists t_2 \leqslant t{:}wait\ to\ re_k?\ during\ (t_2, t_2 + v_k) \\
\Rightarrow \{\text{calculus}\} & \exists k \exists t_2 < \infty{:}wait\ to\ re_k?\ during\ (t_2, t_2 + v_k) \\
\Rightarrow (\text{lemma 5.9}\} & \exists k \exists t_2 < \infty{:}no\{re_k!, re_k\}\ during\ (t_2, t_2 + v_k) \\
\Rightarrow \{C^{P_k}\} & (\exists k{:}error_k)
\end{array}$$

Next we try to prove $(\exists k{:}error_k) \rightarrow (\exists t < \infty{:}wait\ to\ al!\ at\ t\ until\ comm)$.

From $\exists k{:}error_k$ we obtain, by C^{P_k}, $\exists k \exists t_0 < \infty{:}no\{re_k!, re_k\}\ during\ (t_0, t_0 + v_k)$, and thus $\exists k \exists t_0 < \infty{:}no\ comm\ via\ re_k\ during\ (t_0, t_0 + v_k)$. With the current specification of W, however, nothing can be derived from this. The specification of W only expresses how W should behave if it does something on any of the channels. But then W need not do anything; even the simple program **skip** would satisfy its specification. Therefore we modify the specification for W as follows: $C_1^W \wedge C_2^W{:}\{time = 0\}\ W\{true\}$, with

$$\begin{aligned}
C_1^W \equiv\ & \forall t_1 < \infty{:}wait\ to\ al!\ at\ t_1\ until\ comm \rightarrow \\
& \quad \exists k \exists t_2 \leqslant t_1{:}wait\ to\ re_k?\ during\ (t_2, t_2 + v_k), \\
C_2^W \equiv\ & \forall t_3 < \infty{:}no\ comm\ via\ re_k\ during\ (t_3, t_3 + v_k) \rightarrow \\
& \quad \exists t_4 \leqslant t_3 + v_k + K{:}wait\ to\ al!\ at\ t_4\ until\ comm.
\end{aligned}$$

Note that C_0^W follows from C_2^W, because $wait\ to\ re_k?\ during\ (t_0, t_0 + v_k)$ implies by lemma 5.9, $no\ \{re_k!, re_k\}\ during\ (t_0, t_0 + v_k)$, and hence $no\ comm\ via\ re_k\ during\ (t_0, t_0 + v_k)$. Now the proof proceeds as follows, for all k,

$$\begin{array}{ll}
 & (\exists k{:}error_k) \\
\Rightarrow \{C^{P_i}\} & \exists k \exists t_0 < \infty{:}no\{re_k!, re_k\}\ during\ (t_0, t_0 + v_k) \\
\Rightarrow \{\text{definition}\} & \exists k \exists t_0 < \infty{:}no\ \ comm\ via\, re_k\ during\ (t_0, t_0 + v_k) \\
\Rightarrow \{C_2^W\} & \exists k \exists t_0 < \infty\, \exists t_4 \leqslant t_0 + v_k + K{:}wait\ to\ al!\ at\ t_4\ until\ comm \\
\Rightarrow \{\text{calculus}\} & \exists t_4 < \infty{:}wait\ to\ al!\ at\ t_4\ until\ comm.
\end{array}$$

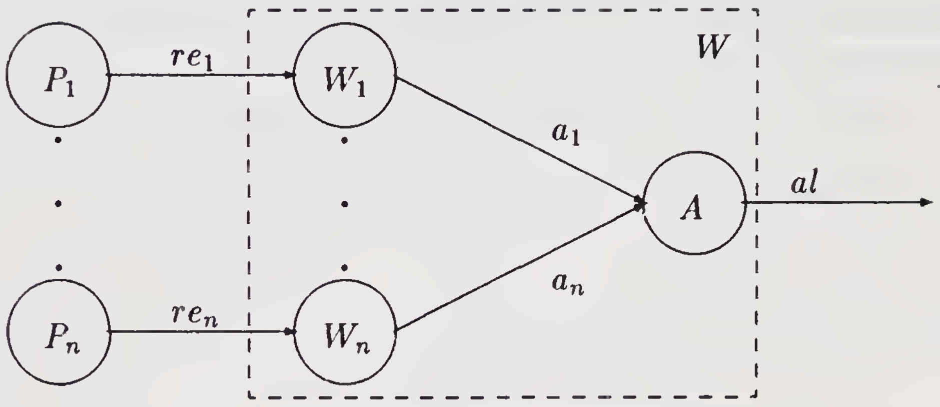


Figure 2. Implementation of the watchdog timer.

6.2 *Implementing the watchdog timer*

Next we design a program implementing watchdog process W that satisfies the required specification. Since W has to watch all processes $P_1, \ldots, P_n$ simultaneously, our first design step is to implement W as a parallel composition, $W \equiv W_1 \| \ldots \| W_n \| A$.

Process W_i is a watchdog for P_i, and it signals process A via channel a_i as soon as there is no communication on re_i for at least v_i time units. Process A waits for a signal on any of the a_i's; after receipt of a signal it tries to send a message on *al* (see figure 2). We give specifications for W_i and A and prove that they are sufficient to derive the specification of W. The specification for W_i expresses that W_i tries to communicate via a_i only if it has been waiting to communicate via re_i during a period of v_i time units. On the other hand, if there is a period of v_i time units during which no communication via re_i occurs, then W_i will try to communicate via a_i within a certain time bound K_i. Define

$$C_1^{W_i} \equiv \forall t_1 < \infty : \textit{wait to } a_i! \textit{ at } t_1 \textit{ until comm} \rightarrow$$
$$\exists t_2 \leqslant t_1 : \textit{wait to } re_i? \textit{ during } (t_2, t_2 + v_i),$$
$$C_2^{W_i} \equiv \forall t_1 < \infty : \textit{no comm via } re_i \textit{ during } (t_3, t_3 + v_i) \rightarrow$$
$$\exists t_4 \leqslant t_3 + v_i + K_i : \textit{wait to } a_i! \textit{ at } t_4 \textit{ until comm.}$$

Then W_i is specified by $C_1^{W_i} \wedge C_2^{W_i} : \{time = 0\}\, W_i \{true\}$.

The specification for A asserts that it tries to send a message via *al* only if there was a preceding communication via one of the a_k. If A is not waiting to communicate via one of the a_k at a certain point of time, then within, say, K_A time units it will wait to communicate via *al* until the actual communication can be performed. Define

$$C_1^A \equiv \forall t_1 < \infty : \textit{wait to } al! \textit{ at } t_1 \textit{ until comm} \rightarrow$$
$$\exists k \exists t_2 \leqslant : \textit{comm via } a_k \textit{ during } [t_2, t_2 + K_c),$$
$$C_2^A \equiv \forall t_3 < \infty : \neg \textit{wait to } a_k? \textit{ at } t_3 \rightarrow$$
$$\exists t_4 < t_3 + K_A : \textit{wait to } al! \textit{ at } t_4 \textit{ until comm.}$$

Then $C_1^A \wedge C_2^A : \{time = 0\}\, A \{true\}$.

We show that $W_1 \| \ldots \| W_n \| A$ satisfies the specification of W (using the specifications of $W_1, \ldots, W_n$, and A only). By the repeated application of the simple parallel

composition rule, we obtain the conjunction of the commitments of the processes: $\bigwedge_{i=1}^{n}(C_1^{W_i} \wedge C_2^{W_i}) \wedge C_1^A \wedge C_2^A$. By the well-formedness axiom and the conjunction rule we can add $Well\ Form_{\{a_k, a_k!, a_k?|k=1,\dots,n\}}$, leading to the following commitment: $\bigwedge_{i=1}^{n}(C_1^{W_i} \wedge C_2^{W_i}) \wedge C_1^A \wedge C_2^A \wedge Well\ Form_{\{a_k, a_k!, a_k?|k=1,\dots,n\}}$.
This implies C_1^W as follows, for all $t_1 < \infty$,

$$\begin{array}{ll}
 & wait\ to\ al!\ at\ t_1\ until\ comm \\
\Rightarrow\{C_1^A\} & \exists k \exists t_2 \leqslant t_1{:}comm\ via\ a_k\ during\ [t_2, t_2 + K_c) \\
\Rightarrow\{\text{definition}\} & \exists k \exists t_2 \leqslant t_1{:}wait\ to\ a_k!\ at\ t_2\ until\ comm \\
\Rightarrow\{C_1^{W_k}\} & \exists k \exists t_2 \leqslant t_1 \exists t_3 \leqslant t_2{:}wait\ to\ re_k?\ during\ (t_3, t_3 + v_k). \\
\Rightarrow\{t_3 \leqslant t_2 \leqslant t_1\} & \exists k \exists t_3 \leqslant t_1{:}wait\ to\ re_k?\ during\ (t_3, t_3 + v_k).
\end{array}$$

Next we prove C_2^W. For all $t_3 < \infty$,

$$\begin{array}{ll}
 & no\ comm\ via\ re_k\ during\ (t_3, t_3 + v_k) \\
\Rightarrow\{C_2^{W_k}\} & \exists t_4 \leqslant t_3 + v_k + K_k{:}wait\ to\ a_k!\ at\ t_4\ until\ comm \\
\Rightarrow\{\text{lemma 5.10}\} & \exists t_4 \leqslant t_3 + v_k + K_k \forall t_5, t_4 < t_5 < t_4 + K_c{:}\neg\, wait\ to\ a_k?\ at\ t_5 \\
\Rightarrow\{C_2^A\} & \exists t_4 \leqslant t_3 + v_k + K_k \forall t_5,\ t_4 < t_5 < t_4 + K_c \exists t_6 < t_5 + K_A{:} \\
 & wait\ to\ al!\ at\ t_6\ until\ comm \\
\Rightarrow\{\text{calculus}\} & \exists t_4 \leqslant t_3 + v_k + K_k \exists t_6 \leqslant t_4 + K_A{:}wait\ to\ al!\ at\ t_6\ until\ comm \\
\Rightarrow\{\text{calculus}\} & \exists t_6 \leqslant t_3 + v_k + K_k + K_A{:}wait\ to\ al!\ at\ t_6\ until\ comm.
\end{array}$$

Hence the specification of W can be derived provided $K_k + K_A \leqslant K$, for all k.

6.3 *Final implementations*

Finally, we give implementations for the processes A and W_i, and we show that these programs meet the required specifications.

Implementation of A: First we show that A can be implemented as $[\square_{i=1}^{n} a_i? \rightarrow al!]$.

We have to prove

$$C_1^A \wedge C_2^A{:}\{time = 0\}[\square_{i=1}^{n} a_i? \rightarrow al!]\{true\}.$$

Define, for $A \equiv [\square_{i=1}^{n} a_i? \rightarrow al!]$ and $i \in \{1, \dots, n\}$,

$C_i^1 \equiv wait\ in\ A\ during\ [t_0, t) \wedge comm\ via\ a_i\ during\ [t, t + K_c)$, and

$C_i^2 \equiv wait\ to\ al!\ at\ t + K_c\ until\ comm.$

We apply the rule for guarded command without delay using

$C_i \equiv \exists t, t_0 \leqslant t < \infty{:}C_i^1 \wedge C_i^2$, $C_{nonterm} \equiv wait\ in\ A\ during\ [t_0, \infty)$,

$p_i \equiv \exists t, t_0 \leqslant t < \infty{:}C_i^1 \wedge time = t + K_c$, and $q_i \equiv true$. Then

1. $wait\ in\ A\ during\ [t_0, \infty) \wedge time = \infty \rightarrow C_{nonterm}$.

2. $\exists t, t_0 \leqslant t < \infty$: *wait in A during* $[t_0, t) \wedge$
   comm via a_i *during* $[t, t+K_c) \wedge time = t + K_c \rightarrow p_i$, for $i \in \{1, \ldots, n\}$.
3. C_i:$\{p_i\}$ *al*! $(q_i\}$, for $i \in \{1, \ldots, n\}$ can be derived as follows.

By the send axiom and the consequence rule,

$$\textit{wait to al! at } t_1 \textit{ until comm}: \{time = t_1\}\ al!\ \{true\}.$$

Applying the adaptation rule, with $exp \equiv t_1 + K_c$, we obtain

$$(p_i[t_1 + K_c/time] \wedge \textit{wait to al! at } t_1 + K_c \textit{ until comm}):$$
$$\{p_i[t_i + K_c/time] \wedge time = t_1 + K_c\}\ al!\ \{true\}.$$

Since $(p_i[t_1 + K_c/time] \wedge \textit{wait to al! at } t_1 + K_c \textit{ until comm}) \rightarrow$
$(\exists t, t_0 \leqslant t < \infty : C_i^1 \wedge t_1 + K_c = t + K_c \wedge \textit{wait to al! at } t_1 + K_c \textit{ until comm}) \rightarrow$
$(\exists t, t_0 \leqslant t < \infty : C_i^1 \wedge C_i^2)$, the consequence rule leads to

$$C_i : \{p_i[t_1 + K_c/time] \wedge time = t_1 + K_c\}\ al!\ \{true\}.$$

By the substitution rule, using $exp \equiv 0$,

$$C_{nonterm}[0/t_0] \vee \bigvee_{i=1}^{n} C_i[0/t_0] : \{time = 0\} \equiv [\square_{i=1}^{n} a_i? \rightarrow al!]\{true\}.$$

Since $p_i \rightarrow \exists t_1 : p_i[t_1 + K_c/time] \wedge time = t_1 + K_c$, by the consequence rule,

$$C_i : \{p_i\}\ al!\ \{q_i\}.$$

Then by the rule for guarded command without delay we obtain

$$C_{nonterm} \vee \bigvee_{i=1}^{n} C_i : \{time = t_0\}[\square_{i=1}^{n} a_i? \rightarrow al!]\{true\}.$$

By the substitution rule, using $exp \equiv 0$,

$$C_{nonterm}[0/t_0] \vee \bigvee_{i=1}^{n} C_i[0/t_0] : \{time = 0\}[\square_{i=1}^{n} a_i? \rightarrow al!]\{true\}.$$

We prove that this commitment implies $C_1^A \wedge C_2^A$.

- First prove $C_1^A \equiv \forall t_1 < \infty$: *wait to al! at* t_1 *until comm* $\rightarrow$
  $\exists k \exists t_2 \leqslant t_1$: *comm via* a_k *during* $[t_2, t_2 + K_c)$
  - By the definition of *wait in A during* $[0, \infty)$, $C_{nonterm}[0/t_0]$ leads to *no* $\{al!, al\}$ *during* $[0, \infty)$, and thus $\forall t_1 < \infty : \neg$ *wait to al! at* t_1 *until comm.*
  - $\bigvee_{i=1}^{n} C_i[0/t_0]$ implies $\exists k \exists t < \infty : C_k^1[0/t_0]$, and thus, by the definition of *wait in A during* $[0, t)$, $\exists k \exists t < \infty$: *no* $\{al!, al\}$ *during* $[0, t) \wedge$ *comm via* a_k *during* $[t, t + K_c)$. Thus, for all $t_1 < \infty$, *wait to al! at* t_1 *until comm* implies $t_1 \geqslant t$, and hence $\exists t \leqslant t_1$: *comm via* a_k *during* $[t, t + K_c)$.
- Next we prove C_2^A, that is,
  $\forall t_3 < \infty : \neg$ *wait to* a_k? *at* $t_3 \rightarrow \exists t_4 < t_3 + K_A$: *wait to al! at* t_4 *until comm*
  - From $C_{nonterm}[0/t_0]$, we obtain $\forall i \in \{1, \ldots, n\}$: *wait to* a_i? *during* $[0, \infty)$, and hence $\forall t_3 < \infty \forall k \in \{1, \ldots, n\}$: *wait to* a_k? *at* t_3.
  - Assume $\neg$ *wait to* a_k? *at* t_3, for $t_3 < \infty$. By $C_k[0/t_0]$, there exists a $t < \infty$ such that *wait to* a_k? *during* $[0, t)$ and *wait to al! at* $t + K_c$ *until comm.* Then $t \leqslant t_3$, thus $t + K_c \leqslant t_3 + K_c$, and hence $\exists t_4 \leqslant t_3 + K_A$: *wait to al! at* t_4 *until comm.* This leads to C_2^A, provided $K_c < K_A$.

Implementation of W_i: Next we implement W_i by $*[re_i? \to \textbf{skip} \,[]\, \textbf{delay } v_i \to a_i!]$ and show that, under certain restrictions, this program satisfies the required specification $C_1^{W_i} \wedge C_2^{W_i}$:$\{time = 0\}\, W_i \{true\}$. Define

$$C_1(t_1) \equiv \textit{wait to } a_i! \textit{ at } t_1 \textit{ until comm} \to$$

$$\exists t_2 \leqslant t_1 : \textit{wait to } re_i? \textit{ during } (t_2, t_2 + v_i),$$

$$C_2(t_3) \equiv \textit{no comm via } re_i \textit{ during } (t_3, t_3 + v_i) \to$$

$$\exists t_4 \leqslant t_3 + v_i + K_i : \textit{wait to } a_i! \textit{ at } t_4 \textit{ until comm.}$$

Then

$$C_1^{W_i} \equiv \forall t_1 < \infty : C_1(t_1) \text{ and } C_2^{W_i} \equiv \forall t_3 < \infty : C_2(t_3).$$

We apply the iteration rule with

$$C_{nonterm} \equiv C_1^{W_i} \wedge C_2^{W_i}, \text{ and } C \equiv \forall t_1 < time : C_1(t_1) \wedge \forall t_3 < time : C_2(t_3).$$

If the assumptions for the iteration rule are fulfilled – which is shown below – we obtain

$$C_{nonterm} \wedge time = \infty : \{C\} * [re_i? \to \textbf{skip} \,[]\, \textbf{delay } v_i \to a_i!] \{false\}.$$

Since $C_{nonterm} \wedge time = \infty \to C_1^{W_i} \wedge C_2^{W_i}$, $time = 0 \to C$, and $false \to true$, the consequence rule leads to $C_1^{W_i} \wedge C_2^{W_i}$:$\{time = 0\}\, W_i \{true\}$.
To apply the iteration rule we have to prove

$$(\forall t < \infty \exists t_0 > t : C[t_0/time]) \to C_{nonterm}, \tag{5}$$

$$C : \{C\} [re_i? \to \textbf{skip} \,[]\, \textbf{delay } v_i \to a_i!] \{C\}. \tag{6}$$

Proof of (5). Observe that $(\forall t < \infty \exists t_0 > t : C[t_0/time]) \equiv$

$$(\forall t < \infty \exists t_0 > t : (\forall t_1 < t_0 : C_1(t_1) \wedge \forall t_3 < t_0 : C_2(t_3))) \to$$

$(\forall t_1 < \infty : C_1(t_1) \wedge \forall t_3 < \infty : C_2(t_3)) \equiv C_{nonterm}$, and hence (5) holds.

Proof of (6). Consider $C^\alpha \equiv \forall t_1 < t_5 : C_1(t_1) \wedge \forall t_3 < t_5 : C_2(t_3)$, and $C^\beta \equiv \forall t_1, t_5 \leqslant t_1 < time : C_1(t_1) \wedge \forall t_3, t_5 \leqslant t_3 < time : C_2(t_3)$. Then $C \leftrightarrow C^\alpha \wedge C^\beta$.
Below we derive

$$C^\beta : \{time = t_5\} [re_i? \to \textbf{skip} \,[]\, \textbf{delay } v_i \to a_i!] \{C^\beta\}. \tag{7}$$

From (7) we obtain by the adaptation rule, with $p \equiv C^\alpha$ and $exp \equiv t_5$,

$$C^\alpha \wedge C^\beta : \{C^\alpha \wedge time = t_5\} [re_i? \to \textbf{skip} \,[]\, \textbf{delay } v_i \to a_i!] \{C^\alpha \wedge C^\beta\}.$$

Using $C \leftrightarrow (C^\alpha \wedge C^\beta)$, by the consequence rule,

$$C : \{C^\alpha \wedge time = t_5\} [re_i? \to \textbf{skip} \,[]\, \textbf{delay } v_i \to a_i!] \{C\}.$$

Since $C \to (\exists t_5 : C[t_5 time] \wedge time = t_5) \to (\exists t_5 : C^\alpha \wedge time = t_5)$, the quantifiction rule and the consequence rule lead to (6).

Proof of (7). Let $G \equiv [re_i? \rightarrow \mathbf{skip} \,[]\, \mathbf{delay}\ v_i \rightarrow a_i!]$.
Apply the rule for guarded command with delay, using
$p_1 \equiv \exists t, t_5 \leqslant t < t_5 + v_i$: *wait to* $re_i?$ *during* $[t_5, t) \wedge$ *comm via* re_i *during* $[t, t+K_c) \wedge$ *no* $\{a_i, a_i!\}$ *during* $[t_5, t+K_c) \wedge time = t + K_c$.
Then $\exists t, t_5 \leqslant t < t_5 + v_i$: *wait in G during* $[t_5, t) \wedge$ *comm* re_i *in G from* $t \rightarrow p_1$, thus we can derive (7), provided

$$C^\beta : \{p_1\} \mathbf{skip} \{C^\beta\}, \tag{8}$$

$$C^\beta : \{\textit{wait in G during } [t_5, t_5 + v_i) \wedge time = t_5 + v_i\} a_i! \{C^\beta\}. \tag{9}$$

Proof of (8). Derive by the skip axiom $time = t_0 : \{time = t_0\} \mathbf{skip} \{time = t_0\}$.
Then by the adaptation rule, with $exp \equiv t + K_c$ and
$p \equiv t_5 \leqslant t < t_5 + v_i \wedge$ *wait to* $re_i?$ *during* $[t_5, t) \wedge$ *comm via* re_i *during* $[t, t+K_c) \wedge$ *no* $\{a_i, a_i!\}$ *during* $[t_5, t+K_c)$,
we obtain $p \wedge time = t + K_c : \{p \wedge time = t + K_c\} \mathbf{skip} \{p \wedge time = t + K_c\}$.
By the consequence rule and the quantification rule we obtain $p_1 : \{p_1\} \mathbf{skip} \{p_1\}$.
Observe that

$$\begin{aligned} p_1 &\Rightarrow \forall t_1, t_5 \leqslant t_1 < time : \neg \textit{ wait to } a_i! \textit{ at } t_1 \wedge \neg \textit{ comm via } a_i \textit{ at } t_1 \\ &\Rightarrow \forall t_1, t_5 \leqslant t_1 < time : \neg \textit{ wait to } a_i! \textit{ at } t_1 \textit{ until comm} \\ &\Rightarrow \forall t_1, t_5 \leqslant t_1 < time : C_1(t_1), \end{aligned}$$

and

$$\begin{aligned} p_1 &\Rightarrow \exists t, t_5 \leqslant t < t_5 + v_i : \textit{ comm via } re_i \textit{ during } [t, t+K_c) \wedge time = t + K_c \\ &\Rightarrow \forall t_3, t_5 \leqslant t_3 < time\ \exists t, t_3 \leqslant t < t_3 + v_i : \textit{comm via } re_i \textit{ at } t \\ &\Rightarrow \forall t_3, t_5 \leqslant t_3 < time : \neg \textit{ no comm via } re_i \textit{ during } (t_3, t_3 + v_i) \\ &\Rightarrow \forall t_3, t_5 \leqslant t_3 < time : C_2(t_3). \end{aligned}$$

Hence $p_1 \rightarrow C^\beta$, and thus the consequence rule leads to (8).

Proof of (9). Define
$C^a \equiv \exists t \geqslant t_5 + v_i$: *wait to* $a_i!$ *at* $t_5 + v_i$ *until comm at* $t \wedge time = t + K_c$.
From the send rule, the conseqence rule and the substitution rule (replacing t_0 by $t_5 + v_i$) we obtain $C^a : \{time = t_5 + v_i\} a_i! \{C^a \wedge time < \infty\}$.
Define $C^b \equiv$ *wait to* $re_i?$ *during* $[t_5, t_5 + v_i) \wedge$ *no* $\{a_i, a_i!\}$ *during* $[t_5, t_5 + v_i)$.
Then by the adaptation rule we can derive

$$C^a \wedge C^b : \{time = t_5 + v_i \wedge C^b\} a_i! \{C^a \wedge C^b \wedge time < \infty\}.$$

Since *wait in G during* $[t_5, t_5 + v_i) \wedge time = t_5 + v_i \rightarrow time = t_5 + v_i \wedge C^b$, we obtain (9) by the consequence rule, if $C^a \wedge C^b$ implies C^β. Recall that

$$\begin{aligned} C^\beta \equiv (&\forall t_1, t_5 \leqslant t_1 < time : \textit{wait to } a_i! \textit{ at } t_1 \textit{ until comm} \rightarrow \\ &\qquad \exists t_2 \leqslant t_1 : \textit{wait to } re_i? \textit{ during } (t_2, t_2 + v_i)) \wedge \\ (&\forall t_3, t_5 \leqslant t_3 < time : \textit{no comm via } re_i \textit{ during } (t_3, t_3 + v_i) \rightarrow \\ &\qquad \exists t_4 \leqslant t_3 + v_i + K_i : \textit{wait to } a_i! \textit{ at } t_4 \textit{ until comm}). \end{aligned}$$

It remains to prove $C^a \wedge C^b \rightarrow C^\beta$:

- For all $t_1, t_5 \leqslant t_1 < time$: if *wait to* a_i! *at* t_1 *until comm*, then from $C^a \wedge C^b, t_1 \geqslant t_5 + v_i$. Hence, from C^b, there exists a $t_2 < t_1$ (viz., t_5) such that *wait to* re_i? *during* $(t_2, t_2 + v_i)$, provided $v_i > 0$.
- Assume, for all t_3, $t_5 \leqslant t_3 < time$: *no comm via* re_i *during* $(t_3, t_3 + v_i)$. From C^a we obtain *wait to* a_i! *at* $t_5 + v_i$ *until comm*). Hence, there exists a $t_4 \leqslant t_3 + v_i + K_i$ such that *wait to* a_i! *at* t_4 *until comm*) if $t_5 + v_i \leqslant t_3 + v_i + K_i$. Since $t_5 \leqslant t_3$, this holds provided $K_i \geqslant 0$.

Conclusion: By giving programs that implement A and W_i, we have obtained an implementation that satisfies the top-level specification for a watchdog timer as given in § 6.1. To conclude this example, we analyse the requirements which have been imposed upon the constants K, K_i, K_A and v_i to prove the correctness of our implementation. To justify the refinement step from the previous section, we required $K_i + K_A \leqslant K$, for all $i = 1, \ldots, n$. The implementation given in this section has been proved to satisfy the specification for all K_A and K_i such that $K_c < K_A$ and $K_i \geqslant 0$, and provided $v_i > 0$, for all $i = 1, \ldots, n$. Observe that if $K > K_c$ then for $K_A = K$ and $K_i = 0$ we have $K_c < K_A$ and $K_i \geqslant 0$, and also $K_i + K_A = K \leqslant K$. Hence, if $v_i > 0$ for all $i = 1, \ldots, n$, and $K > K_c$, i.e., the constant in the specification must be greater than the duration of a communication, then our implementation meets the top-level specification for W.

7. Adding assumptions to real-time

In general, real-time embedded systems have an intensive interaction with their environment, and usually their correctness strongly depends on assumptions about the environment. Therefore it is convenient to use correctness formulae in which these assumptions can be expressed. Hence, similar to § 4, we extend the formulae $C\colon\{p\}S\{q\}$ from § 5 with a fourth assertion, called *assumption*, leading to formulae of the form $(A, C)\colon\{p\}S\{q\}$. Assumption A should neither contain program variables nor the special variable *time*.

In this assumption/commitment formalism for real-time properties, we can now express, for instance, in the assumption when the environment of a process is waiting to communicate. With such an assumption we can determine when the communication must take place. For example,

$$(A \equiv \textit{wait to } c! \textit{ at } 5 \textit{ until comm} \wedge \textit{no comm via } c \textit{ during } [3, 5),$$

$$C \equiv \textit{comm via } c \textit{ during } [5, 5 + K_c)):$$

$$\{time = 3\}\, c?\, \{time = 5 + K_c\}.$$

Note that, by using the maximal parallelism model, a communication takes place as soon as both process and environment are ready to perform the communication.

In the remainder of this section we indicate how the proof system from § 5 can be extended to obtain a compositional proof system for these assumption/commitment-formulae. First we consider a compositional rule for the parallel composition of S_1

and S_2. Concerning pre-conditions, post-conditions and commitments, the rule is identical to rule 5.27 for the commitment-formalism. For the assumptions, we have same requirements as in rule 4.2: $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$.

Rule 7.1. (Parallel composition)

$$\frac{\begin{array}{l}(A_1, C_1){:}\{p_1\}S_1\{q_1\},\ (A_2, C_2){:}\{p_2\}S_2\{q_2\}\\ q_1[t_1/time] \wedge q_2[t_2/time] \wedge time = max(t_1, t_2) \rightarrow q\\ C_1[t_1/time] \wedge C_2[t_2/time] \wedge time = max(t_1, t_2) \rightarrow C\\ A \wedge C_1 \rightarrow A_2,\ A \wedge C_2 \rightarrow A_1\end{array}}{(A, C){:}\{p_1 \wedge p_2\}S_1 \| S_2\{q\}},$$

provided t_1 and t_2 are fresh logical variables, and $dch(C_i, q_i) \subseteq dch(S_i)$, for $i \in \{1, 2\}$. A typical application of this rule can be found in the next example.

Example 7.1. Consider the following specifications (**delay** d is used to represent any internal actions which takes d time units and assume communications take one time unit, i.e., $K_c = 1$):

$(A_1 \equiv$ *wait to c? at* 2 *until comm* $\wedge$ *no comm via c during* $[0, 2) \wedge$

wait to d? at 6 *until comm* $\wedge$ *no comm via d during* $[3, 6)$,

$(C_1 \equiv$ *comm via c during* $[2, 3) \wedge$ *wait to d! at* 3 *until comm* $\wedge$

no comm via d during $[0, 3))$:

$\{time = 0\}$ c!; *d*!; **delay** $2\{time = 9\}$, and

$(A_2 \equiv$ *wait to d! at* 3 *until comm* $\wedge$ *no comm via d during* $[0, 3)$,

$C_2 \equiv$ *wait to d? at* 6 *until comm* $\wedge$ *no comm via d during* $[0, 6) \wedge$

comm via d during $[6, 7))$:

$\{time = 0\}$ **delay** 6; *d*? $\{time = 7\}$.

Take for (c!; *d*!; **delay** 2) $\|$ (**delay** 6; *d*?) the following assumption:

$A \equiv$ *wait to c? at* 2 *until comm* $\wedge$ *no comm via c during* $[0, 2)$.

Since $A \wedge C_1 \rightarrow A_2$ and $A \wedge C_2 \rightarrow A_1$, the parallel composition rule leads to

$(A, C_1 \wedge C_2){:}\{time = 0\}$ (c!; *d*!; **delay** 2) $\|$ (**delay** 6; *d*?) $\{time = 9\}$.

Using a consequence rule we can easily derive from $C_1 \wedge C_2$ the following commitment: *comm via c during* $[2, 3) \wedge$ *comm via d during* $[6, 7)$. □

Similar to section 4, to achieve a sound rule for parallel composition, an inductive relation between assumption and commitment is necessary to avoid circularity. In our real-time specifications we require for the validity of $(A, C){:}\{p\}S\{q\}$ that there exists a $\delta > 0$ such that

1. for all t with $0 \leqslant t < \delta$: C holds at t, and
2. for all $t \geqslant \delta$: if A holds at $t - \delta$ then C holds at t.

In our examples this requirement is fulfilled if *comm via D during* $[t_0, t_1)$ is not considered as an abbreviation but as a primitive which is trivially true at all points of time before t_1.

Other rules and axioms for the assumption/commitment formalism can be obtained by adapting the commitment-based proof system of the previous section. We simply add assumption *true* to the rule and axioms for atomic statements, and the proof system is extended by a rule that allows the addition of an assumption to strengthen commitment and post-condition. To formulate this rule, let $p@t$ denote assertion p at time t, ignoring the part of p that refers to point of time after t. (By definition, $(p@t) \equiv true$ if $t < 0$.) Then, for all $\delta > 0$, we have the following rule.

Rule 7.2. (Strengthen)

$$\frac{\begin{array}{l}(A_1, C_1){:}\{p_1\}S\{q_1\}\\ \forall t{:}(A@(t-\delta)) \wedge (C_1@t) \rightarrow (C@t)\\ \forall t{:}(A@(t-\delta)) \wedge (q_1@t) \rightarrow (q@t)\end{array}}{(A_1 \wedge A, C){:}\{p_1\}S\{p_1\}}.$$

Furthermore, the rules for compound statements have to be adapted, and in the consequence rule we require that all implications hold point-wise.

Rule 7.3. (Consequence)

$$\frac{\begin{array}{l}(A_1, C_1){:}\{p_1\}S\{q_1\}\\ \forall t{:}(A@t) \rightarrow (A_1@t),\ \forall t{:}(p@t) \wedge (time < \infty) \rightarrow (p_1@t)\\ \forall t{:}(C_1@t) \rightarrow (C@t),\ \forall t{:}(q_1@t) \rightarrow (q@t)\end{array}}{(A, C){:}\ \{p\}S\{q\}}.$$

8. Related work and state of the art

For concurrent programs communicating via message passing as well as for shared variable concurrency, one can observe a development from non-compositional proof methods which require the (final) program text for their application, such as Owicki & Gries (1976), Apt *et al* (1980) and Levin & Gries (1981), towards compositional theories, e.g. Chen & Hoare (1981, pp. 1–12), Soundararajan (1984b), Stirling (1986, pp. 407–415), Zwiers (1989) and Stølen (1990) (see de Roever 1985b, pp. 181–207, and Hooman *et al* 1986 for an overview of this development). An early Indian pioneer in compositional proof methods for concurrency is Soundararajan (Soundararajan 1984; Sobel & Soundararajan 1985, pp. 343–359). Whereas these methods verify only safety properties, with temporal logic (Pnueli 1977, pp. 46–57; Manna & Pnueli 1982, pp. 163–255) also liveness (progress) properties can be verified. Compositional proof systems for temporal logic have been given in Barringer *et al* (1984, pp. 51–63) and Nyugen *et al* (1986). In Pandya & Joseph (1991) a compositional proof system called P-A logic (for presupposition–affirmation logic) is described for establishing weak total correctness and weak divergence correctness of CSP-like distributed programs with synchronous and asynchronous communication. This extension allows compositional deadlock proofs and, moreover, compositional proof rules are given for *until*-properties of the form *Q until R*, where *Q* and *R* are assertions over communication traces. It seems that this paper describes how far one can go towards proving liveness in a compositional framework using a non-temporal formalism in an assumption-commitment based setting.

Interestingly, more involved programming language fragments for concurrency, such as the concurrency fragment of Ada (1983) or those for monitor based languages, have not been characterized until now through compositional trace-based methods, although their non-compositional characterization was possible and has been given – see de Roever (1985a, pp. 213–260) for an overview. However, if one allows locations inside programs as observables, there exists a straightforward technique to convert non-compositional proof methods for concurrency to compositional ones, as reported in Gerth & de Roever (1986). Similarly, Stirling (1986, pp. 407–416) reports how a basically non-compositional proof method for shared variable concurrency, such as the one by Owicki & Gries (1976), can be reformulated compositionally within a framework based on relevance logic. Also for the parallel object oriented language POOL first non-compositional proof methods (America & de Boer 1990) have been developed based on the method of Apt *et al* (1980). The principal author, de Boer, recently reformulated his proof system along history-based compositional lines (de Boer 1991) using the work of Zwiers (1989) as a starting point.

In the present paper we discuss a compositional proof system for distributed message passing in which assumptions can be made about the behaviour of the environment in the style of Misra & Chandy (1981) and Zwiers *et al* (1984). The main idea of the method is that suitable assumptions about the environment reduce the immence number of possible behaviours of complex reactive systems. Misra & Chandy (1981) were the first ones to demonstrate the advantages of assumptions in the hierarchical design and verification of distributed processes with message passing. They proposed a compositional rule for the parallel operator and demonstrated their method on several examples. These ideas have been formalized by Zwiers *et al* (1984), resulting in a compositional proof system for assumption/commitment based specifications together with its soundness and completeness proof. The examples in Ossefort (1983) show that the Misra–Chandy method is easy to use indeed and that it leads to simple and natural correctness proofs. In Pandya (1988) and Pandya & Joseph (1991), the formalism of Zwiers *et al* (1984) is extended to asynchronous communication and progress properties. Also related is the formalism given by Stark (1985, pp. 369–391), who uses rely- and guarantee-conditions for deriving global liveness properties of a distributed system. Interestingly, at present new non-standard applications of the assumption-commitment framework are burgeoning, e.g., for characterizing specifications of fault-tolerant processes or within development methods for mutual exclusion algorithms in which the final algorithm is obtained by a series of error containing approximations in which errors are gradually removed until all are absent, see e.g. Cau & Kuiper (1991). A compositional theory for action refinement is developed by Jannsen *et al* (1991), in a setting of partial orders, and this theory is applied to proving the correctness of distributed databases by formalizing the notion of serializability. It contains an example of how a general specification using an unbounded number of processors is refined to an implementation using two processors. Recently, a number of developments in assumption/commitment based reasoning have taken place; for the state of the art consult the papers by Pandya (1989, pp. 622–640) and Abadi & Lamport (1989, pp. 1–41).

The present paper focusses on concurrent processes with synchronous message passing along channels. What about compositional approaches to shared-variable concurrency and asynchronous message passing? As to shared-variable concurrency, a basic observation is made by Aczel (as reported in de Roever 1985b, pp. 181–207), in which in the semantics of a process a distinction is made between a so-called "component action" Π (an action of the process itself) and an "environment action" E (an update to a shared variable by another process). Then (x, Π) and (x, E) are the analogues of the communication records in the channel-based theory above. The original reference in which these notions where informally introduced is Jones (1981); Jones (1983) is a more accessible reference to these ideas and contains a proposal for assumption/commitment based reasoning (called rely/guarantee reasoning) about shared variable concurrency together with the formulation of a compositional proof rule for the parallel operator. The idea is exemplified in Woodcock & Dickinson (1988) and formally worked out in Stølen (1990). In a mixed (temporal logic)-(transition system) based approach, asynchronous communication is characterized compositionally in the work by Jonsson (1987a, 1987b, pp. 152–166). It is shown (de Boer *et al* 1990) that for a compositional description of any programming language based upon asynchronous communication a trace model is sufficient, i.e., no additional structures to encode some relevant branching information (trees, failure sets) are needed. A compositional axiomatization is given (Hooman *et al* 1990, pp. 242–261, 1992) for the graphical specification language Statecharts which includes features like concurrency, broadcast communication, and time-out.

A dichotomy is observed (Zwiers & Roever 1989) in compositional proof theories for concurrency; one class of methods (including, e.g., temporal logic and VDM), is based on programs as predicates, and has a simple proof theory (due to the power of the consequence rule), but has trouble in characterizing sequential composition and iteration. Methods in the other class (including weakest pre-condition calculi, Hoare triples and dynamic logics), are based on programs as predicate transformers, and have no trouble in dealing with sequential composition and iteration, but are more complicated due to awkward implication rules. An attempt at unification is made (Zwiers & de Roever 1989) by using adjoints.

Except for Statecharts, these methods are not designed to verify and specify real-time properties. Now an obvious approach towards a verification theory for real-time programs is to adapt and extend an already existing method which does not incorporate any notion of time. For instance, in traditional linear temporal logic, safety and liveness properties are expressed by means of a qualitative notion of time (e.g. "eventually", "henceforth", "until"). In order to express real-time constraints, extensions of this logic have been proposed (Bernstein & Harter 1981, pp. 1–11; Shankar & Lam 1987; Koymans 1989) which also includes a quantitative notion of time (e.g. "eventually within 5 time units", "always after 7 time units"). These extensions have been applied to the specification of real-time communication properties of a transmission medium (Koymans *et al* 1983, pp. 187–197) and the verification of local area network protocols (Shasha 1984, pp. 54–65). A compositional proof theory for real-time distributed message passing using an assertion language based on real-time temporal logic has been given in Hooman & Widom (1989, pp. 424–441). In Hooman (1991b) this compositional method is extended to uniprocessor implementations and priorities. Non-compositional proof methods, based on Manna & Pnueli's (1982, pp. 163–255) classical approach to linear time temporal logic, can be found in Harel (1988) and Ostroff (1989). They express real-time properties in explicit clock temporal logic and give decision procedures for this logic.

Similarly, real-time extensions have been formulated for other methods. There is an early paper of Haase (1981) in which time is introduced by a special variable in the weakest pre-condition calculus. Bernstein (1987) discusses several ways of modelling message passing with time-out in the non-compositional framework of Levin & Gries (1981). Zwarico & Lee (1985, pp. 169–177) have adapted Hoare's (1985) trace model (with one invariant and a satisfaction relation) to real-time. Nested parallelism is not allowed in their programming language, a restricted version of sequential composition is used, and there is no explicit mechanism for expressing time constraints. A real-time logic to analyze safety properties is defined (Jahanian & Mok 1986) based on a function which assigns a time value to each occurrence of an event. Real-time properties of sliding window protocols are verified by Shankar & Lam (1987) using special state variables, called timers, to measure the passage of time. The compositional proof system from Davies & Schneider (1989, pp. 129–159) and Schneider (1990) for timed CSP supports semantic reasoning in the framework of Reed & Roscoe (1986, pp. 314–323). Furthermore, Schneider (1990) defines a notion of time-wise refinement to transfer properties of non-timed CSP programs to their timed version, thus exploiting the hierarchy of timed and untimed models from Reed (1989, pp. 80–128). Beaten & Bergestra (1990) have incorporated real-time aspects in the process algebra of Bergestra & Klop (1984) by adding time stamps to atomic actions. In their approach atomic actions have a positive duration, whereas in the process algebra of Nicollin *et al* (1990, pp. 402–429) actions have no duration in general, except a distinguished time action which models the ticks of a synchronized global clock. Furthermore, Nicollin *et al* (1990, pp. 402–429), present a systematic approach to delay-constructs. Milner's (1989) CCS is extended by Moller & Tofts (1990, pp. 401–415) and Yi (1990, pp. 502–520) with explicit time. To obtain a calculus for shared resources, in Gerber & Lee (1990, pp. 263–277) a priority-based process algebra is presented.

This work was partially supported by Esprit-BRA project 3096: Formal Methods and Tools for the Development of Distributed and Real-Time Systems (SPEC).

References

Ada 1983 *The Programming Language Ada* (Reference manual)

America P, de Boer F 1990 A proof system for process-creation. In *TC-2 Working Conference on Programming Concepts and Methods*

Apt K R, Francez N, de Roever W P 1980 *A proof system for Communicating Sequential Process. ACM Trans. Program. Languages Syst.* 2: 359–385

Abadi M, Lamport L 1989 Composing specifications. In *Stepwise refinement of distributed systems* (Berlin: Springer-Verlag)

Baeten J C M, Bergestra J A 1990 Real time process algebra, Technical Report P8916b, University of Amsterdam

Bernstein A J 1987 Predicate transfer and timeout in message passing. *Inf. Process. Lett.* 24: 43–52

Bernstein A, Harter P K Jr 1981 Proving real-time properties of programs with temporal logic. In *Proceedings of the 8th Annual ACM Symposium on Operating System Principles* (New York: ACM Press)

Bergestra J A, Klop J W 1984 Process algebra for synchronous communication. *Info. Control* 60: 109–137

Barringer H, Kuiper R, Pnueli A 1984 Now you may compose temporal logic specifications. In *Proceedings of the 16th Annual Symposium on Theory of Computing* (New York: ACM Press)

Chen Z C, Hoare C A R 1981 Partial correctness of Communicating Sequential Processes. In *Proceedings of the IEEE International Conference on Distributed Computing Systems* (New York: IEEE Press)

Cau A, Kuiper R 1991 Formalising Dijkstra's development strategy using Stark's formalism, Technical Report, Christian-Albrechts-University, Kiel

de Boer F 1991 A compositional proof system for dynamic process creation. In *Proceedings Symposium on Logic in Computer Science*

de Boer F, Kok J, Palamidessi C, Rutten J 1990 The failure of failures: towards a paradigm for asynchronous communication, Technical Report, University of Utrecht

de Roever W P 1985a The cooperation test: a syntax directed verification method. In *Logics and Models of Concurrent Systems. NATO ASI Series F, Vol, 13* (Berlin: Springer-Verlag)

de Roever W P 1985b The quest for compositionality – a survey of assertion-based proof systems for concurrent programs, Part I: concurrency based on shared variables. In *Proceedings of the IFIP Working Conference 1985: The role of abstract models in computer science* (Amsterdam: North-Holland)

Davies J, Schneider S 1989 Factorizing proofs in timed CSP. In *Mathematical foundations of programming semantics. Lecture Notes in Computer Science. Vol. 442* (Berlin: Springer-Verlag)

Gerber R, Lee I 1990 CCSR: a calculus for communicating shared resources. In *CONCOUR*'90. *Lecture Notes in Computer Science. Vol. 458* (Berlin: Springer-Verlag)

Gerth R, de Roever W P 1986 Proving monitors revisited: a first step towards verifying object oriented systems. *Fundam. Inf.* 9: 371–400

Haase V H 1981 Real-time behaviour of programs. *IEEE Trans. Software Eng.* SE-7: 494–501

Harel E 1988 *Temporal analysis of real-time systems*, Master's thesis, The Weizmann Institute of Science, Rehovot, Israel

Hooman J, de Roever W P 1986 The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current trends in concurrency. Lecture Notes in Computer Science. Vol. 224* (Berlin: Springer-Verlag)

Huizing C, Gerth R, de Roever W P 1987 Full abstraction of a real-time denotational semantics for an OCCAM-like language. *In Proceedings of the 14th ACM Symposium on Principles of Programming Languages* (New York: ACM Press)

Hoare C A R 1969 An axiomatic basis for computer programming. *Commun. ACM* 12: 576–580, 583

Hoare C A R 1985 *Communication sequential process* (Englewood Cliffs, NJ: Prentice Hall)

Hooman J 1991 A denotational real-time semantics for shared processors. In *Parallel Architectures and Languages Europe. Lecture Notes in Computer Science. Vol. 506* (Berlin: Springer-Verlag)

Hooman J 1991 *Specification and compositional verification of real-time systems*, Ph D thesis, Eindhoven University of Technology. Also in (1991) *Lecture Notes in Computer Science. Vol. 558* (Berlin: Springer-Verlag)

Harel D, Pnueli A 1985 On the development of reactive systems. In *Logics and models of concurrent systems, NATO, ASI-13* (Berlin: Springer-Verlag)

Hooman J, Ramesh S, de Roever W P 1991 A compositional axiomatization of Statecharts. *Theor. Comput. Sci.* (to appear)

Hooman J, Ramesh S, de Roever W P 1990 A compositional axiomatisation of safety and liveness properties of Statecharts. In *Semantics for concurrency.* Workshop in Computing (Leicester: Springer-Verlag)

Hooman J, Widom J 1989 A temporal-logic based compositional proof system for real-time message passing. In *Parallel Architectures and Languages Europe. Lecture Notes in Computer Science. Vol. 366* (Berlin: Springer-Verlag) vol. 2

Joseph M, Goswami A 1989 Relating computation and time, Research Report RR 138, Department of Computer Science, University of Warwick

Jahanian F, Mok A 1986 Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng.* SE-12: 890–904

Jones C B 1981 *Development methods for computer programs including a notion of interference*, Ph D thesis, Oxford University, Technical Monograph

Jones C B 1983 Tentative steps towards a development method for interfering programs. *ACM Trans. Program. Lang. Syst.* 5: 596–619

Jonsson B 1987a *Compositional verification of distributed systems*, Ph D thesis, Uppsala University

Jonsson B 1987b Modular verification of asynchronous networks. In *Proceedings 6th ACM Symp. Principles Distrib. Computing* (New York: ACM Press)

Janssen W, Poel M, Zwiers J 1991 Action systems and action refinement in the development of parallel systems, Technical Report, Twente University

Kleene S C 1952 *Introduction to metamathematics* (New York: Van Nostrand)

Koymans R 1989 *Specifying message passing and time-critical systems with temporal logic*, Ph D thesis, Eindhoven University of Technology

Koymans R 1990 Specifying real-time properties with metric temporal logic *Real-time Syst.* 2: 255–299

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(3): 210–256

Koymans R, Vytopyl J, de Roever W P 1983 Real-time programming and asynchronous message passing. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing* (New York: ACM Press)

Levin G M, Gries D 1981 A proof technique for communicating sequential processes. *Acta Inf.* 15: 281–302

Misra J, Chandy K M 1981 Proofs of networks of processes. *IEEE Trans. Software Eng.* SE-7: 417–426

Milner R 1989 *Communication and concurrency* (Englewood Cliffs, NJ: Prentice Hall)

Manna Z, Pnueli A 1982 Verification of concurrent programs: a temporal proof system. In *Foundations of Computer Science IV, Distributed Systems, Part 2-Vol. 159. Mathematical Centre Tracts* (Amsterdam: CWI)

Moller F, Tofts C 1990 A temporal calculus of communicating systems. In *CONCUR' 90. Lecture Notes in Computer Science. Vol. 458* (Berlin: Springer-Verlag)

Nguyen V, Demers A, Gries D, Owicki S 1986 A model and temporal proof system for networks of processes. *Distrib. Comput.* 1: 7–25

Nicollin X, Richier J L, Sifakis J, Voiron J 1990 ATP: an algebra for timed processes. In *Proceedings IFIP Working Group Conference on Programming Concepts and Methods*

OCCAM 1988 OCCAM2 Reference Manual, INMOS Limited (New York: Prentice-Hall)

Owicki S, Gries D 1976 An axiomatic proof technique for parallel programs. *Acta Inf.* 6: 319–340

Ossefort M 1983 Correctness proofs of communicating processes: Three illustrative examples from the literature. *ACM Trans. Program. Lang. Syst.* 5: 620–640

Ostroff J 1989 *Temporal logic for real-time systems. Advanced Software Development Series* (New York: John Wiley & Sons)

Pandya P 1988 *Compositional verification of distributed programs*, Technical Report CS-88/3 (Ph D thesis), Tata Institute of Fundamental Research, Bombay

Pandya P 1989 Some comments on the assumption-commitment framework for compositional verification of distributed programs. In *Stepwise refinement of distributed systems. Lecture Notes in Computer Science. Vol. 430* (Berlin: Springer-Verlag)

Pandya P, Joseph M 1991 P-A logic – a compositional proof system for distributed programs. *Distrib. Comput.* 4(4): 37–54

Pnueli A 1977 The temporal logic of programs. In *Proceedings of the 18th Symposium on Foundations of Computer Science*

Reed G M 1989 A hierarchy of domains for real-time distributed computing. In *Mathematical Foundations of Programming Semantics. Lecture Notes in Computer Science. Vol. 442* (Berlin: Springer-Verlag)

Reed G, Roscoe A 1986 A timed model for Communicating Sequential Processes. In *Proceedings of ICALP'86 Lecture Notes in Computer Science. Vol. 226* (Berlin: Springer-Verlag)

Schneider S 1990 *Correctness and communication in real-time systems*, Ph D thesis, Oxford University

Shanker A U, Lam S S 1987 Time-dependent distributed systems: proving safety, liveness and real-time properties. *Distrib. Comput.* 2: 61–79

Soundararajan N 1984 *Axiomatic semantics of communicating sequential processes. ACM Trans. Program. Lang. Syst.* 6: 647–662

Soundararajan N 1984 A proof technique for parallel programs. *Theor. Comput. Sci.* 31: 13–29

Shasha D E, Pnueli A, Ewald W 1984 Temporal verification of carrier-sense local area network protocols. In *Proceedings 11th ACM Symposium on Principles of Programming Languages*

Sobel N, Soundararajan N 1985 A proof system for distributed process. In *Logics of programs. Lecture Notes in Computer Science. Vol. 193* (Berlin: Springer-Verlag)

Stark E 1985 A proof technique for rely/guarantee properties. In *Proceedings 5th Conference on Foundations of Software Technology and Theoretical Computer Science. Lecture Notes in Computer Science. Vol. 206* (Berlin: Springer-Verlag)

Stirling C 1986 A compositional reformulation of Owicki-Gries's partial correctness logic for a concurrent while language. In *Proceedings 13th International Colloquium on Automata, Languages and Programming. Lecture Notes in Computer Science. Vol. 226* (Berlin: Springer-Verlag)

Stφlen K 1990 *Development of parallel programs on shared data-structures.* Ph D thesis, Manchester University

Woodcock J, Dickinson B 1988 Using VDM with rely and guarantee-conditions, experiences from a real project. In *Second VDM-Europe Symposium. Lecture Notes in Computer Science. Vol. 328* (Berlin: Springer-Verlag)

Yi W 1990 Real-time behaviour of asynchronous agents. In *CONCUR'90. Lecture Notes in Computer Science. Vol. 458* (Berlin: Springer-Verlag)

Zwarico A, Lee I 1985 Proving a network of real-time processes correct. In *Proceeding IEEE Real-Time Symposium*

Zwiers J, de Roever W 1989 Predicates are predicate transformers: a unified compositional theory for concurrency. In *Proceeding 8th ACM Symposium on Principles of Distributed Computing* (New York: ACM Press)

Zwiers J, de Roever W P, Van Emde Boas P 1984 Compositionality and concurrent networks: soundness and completeness of a proof system, Technical Report 57, University of Nijmegen

Zwiers J 1989 Compositionality, concurrency and partial correctness. *Lecture Notes in Computer Science. Vol. 321* (Berlin: Springer-Verlag)

Sādhanā, Vol. 17, Part 1, March 1992, pp. 75–93. © Printed in India.

Compositional priority specification in real-time distributed systems

R K SHYAMASUNDAR[§] and L Y LIU[‡]

[§]Computer Science Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, India
[‡]IBM Programming Systems, Cary Laboratory, 11000 Regency Parkway, Cary, NC 27511, USA

Abstract. In this paper, we develop a compositional denotational semantics for prioritized real-time distributed programming languages. One of the interesting features is that it extends the existing compositional theory proposed by Koymans *et al* (1988) for prioritized real-time languages preserving the compositionality of the semantics. The language permits users to define situations in which an action has priority over another action without the requirement of preassigning priorities to actions for partially ordering the alphabet of actions. These features are part of the languages such as Ada designed specifically keeping in view the needs of real-time embedded systems. Further, the approach does not have the restriction of other approaches such as prioritized internal moves can pre-empt unprioritized actions etc. Our notion of priority in the environment is based on the intuition that a low priority action can proceed only if the high priority action cannot proceed due to lack of the handshaking partner at that point of execution. In other words, if some action is possible corresponding to that environment at some point of execution then the action takes place without unnecessary waiting. The proposed semantic theory provides a clear distinction between the semantic model and the execution model – this has enabled us to fully ensure that there is no unnecessary waiting.

Keywords. Compositional specification; real-time distributed systems; priority specification; message passing models.

1. Introduction

Many approaches have been proposed for the modelling of communicating agents (Milner 1980), reactive systems (Pnueli & Harel 1988, pp. 84–98), and real-time distributed systems (Roscoe 1984; Koymans *et al* 1985, 1988). Most of the above studies have ignored the notion of *priority*. This is not satisfactory as priority is very important in the development of predictable systems (Stankovic 1988). Priority specification is required when one or more events happen at the same time and some events have greater importance than others. Typical examples of actions which require special treatment include interrupts in hardware systems and timeouts in communication protocols. Ada and Occam are two examples of programming languages that allow specification of priority. One can broadly distinguish priority specification into the following categories:

(1) Partial order on the alphabet of actions/events.
(2) Priority specification through evaluating expressions dynamically and assigning priorities to actions/statements.
(3) Priority of actions depending on the environment.

Perhaps the first formal study of priorities has been done in the context of process algebra in Baeten *et al* (1985). In this study, prioritization operators have been added and the consistency of the set of equations have been studied. In other words, the study falls into type (1) described above. That is, it assumes a global partial ordering of the actions (of the transition system). From a behavioural equivalence point of view, a study has been made in Cleaveland & Hennessy (1990) through the study of priority operators of type (1) in the context of calculus of communicating systems (CCS). Congruence has been obtained through the notion of *patient processes* by placing essentially the restriction that only prioritized internal actions have the pre-emptive power. The desirability of overcoming this restriction follows from the examples discussed in Cleaveland & Hennessy (1990).

A study of the second type of priority has been made in Hooman (1989) in the context of the study of distributed multi-processing. Here, priorities are attached to statements and the priorities of statements on different processors are incomparable. In other words, the relative ordering of priorities on a single processor determines the execution order; for example, if two synchronized actions have different priorities, then the priority for the synchronized action is given by the minimum of the two. Another way of overcoming this problem is to use *one-way* priority, that is, either the input or output action can be assigned priority but not both.

Let us look at the specification of reactive systems. Reactive systems maintain a continuous interaction with their environment at a speed determined by the environment rather than the program itself. In other words, the outputs may affect future inputs due to feedback. One of the primary goals of the study of reactive (real-time) systems is to develop predictable systems (cf. Stankovic 1988). Thus, it becomes essential to predict the action of each component in the context of nondeterministic interacting environment. For this purpose, priority plays a vital role. Hence, priority of type (3) discussed above plays an important role in distributed reactive systems. Let us look at the priority specification characteristics in Ada which has been designed keeping in view the design of real-time embedded systems. In Ada each task may (but need not) have a priority. We quote below relevant aspects from the revised Ada manual (cf. 9·8) given in Gehani (1983):

- The specification of priority is an indication given to assist the implementation in the allocation of processing resources to parallel tasks when there are more tasks eligible for execution than can be supported simultaneously by the available processing resources. The effect of priorities on scheduling is defined by the following rule:
  If two tasks with different priorities are both eligible for execution and could sensibly be executed using the same physical processors and the same other processing

resources, then it cannot be the case that the task with the lower priority is executing while the task with the higher priority is not.

- The above rule essentially corresponds to the *pragma* feature. The most important aspect related to priority specification in the context of a rendezvous is cited below:
  For tasks of the same priority, the scheduling is not defined in the language.... The priority of the task is static and therefore fixed. However, the priority during a rendezvous is not necessarily static since it also depends on the priority of the task calling the entry.

It can be seen from the above that priority specification in Ada corresponds to that of type (3) described above. In appendix A, we illustrate through an example priority specification in Ada by considering the problem of minimizing head movement during a disc access. Priority specifications corresponding to type (3) given above play an important role in the specification of process-control systems.

A limited treatment of priority of type (3) has been reported in Pitassi *et al* (1986). The study in Pitassi *et al* (1986) corresponds to a special case of the prioritized alternative construct of Occam (cf. Occam 1984). The limitation can be understood by the informal interpretation of the alternative construct given below: Upon entering a prioritized alternate (ALT) statement, a linear sequence of all the open guards of the prioritized statement is constructed; if one or more of the open guards is successful upon entry, then the first successful guard in that sequence is selected and the corresponding statement is executed. If none of the open guards is successful upon entry, then the prioritized ALT construct is treated as if it were an ordinary ALT construct.

So far in the literature, there is no indication as to what approach would be realistic and useful. In this paper, we provide a formal semantics for priority in the context of real-time distributed programming languages that have the feature of specifying priority in an environment rather than providing a global partial order of actions in the system. The main contributions of the paper can be summarized as follows.

- An understanding of the notion of priority in the context of environment (i.e., real-time distributed concurrency) is provided. Our approach does not have the restriction (as in Cleaveland & Hennessy 1990) that only prioritized internal moves can pre-empt unprioritized actions. Our notion of priority in the environment is based on the intuition that a low priority action can proceed only if the high priority action cannot proceed due to lack of the handshaking partner at that point of execution. In other words, if some action is possible corresponding to that environment at some point of execution then the action takes place without unnecessarily waiting. It must be clear that such an approach clearly satisfies the requirement given in Lamport (1985) that realistic priority specification should not involve unnecessary waiting.
- A compositional semantic characterization of real-time distributed languages with priority is presented and, thus, forms a basis for the compositional proof theories of languages such as Ada and Occam.

The rest of the paper is organized as follows: Section 2 describes an abstraction of prioritized real-time distributed concurrency in terms of an extension of real-time communicating sequential processes (CSP-R) (Koymans *et al* 1988) and is referred to as CSP-R$_p$; further, the language syntax and informal interpretation of CSP-R$_p$ is discussed. Section 3 describes the real-time model and the execution model, and § 4 describes the semantic domain and semantic equations. Section 5 discusses parallel composition both informally and formally; towards, the end of § 5, we discuss the impact of various parameters including those that affect maximal parallelism on the prioritized semantics proposed. The paper concludes with a discussion of the results and the ongoing work.

2. Language syntax

For ease of presentation, we use language CSP-R$_p$ (prioritized real-time CSP) instead of Ada. It may be noted that CSP-R$_p$ is an extension of CSP-R described in Koymans *et al* (1988). In Koymans *et al* (1988), it has been shown that Ada tasking and real-time features can be simulated by CSP-R. Further, CSP-R$_p$ has the priority features corresponding to that of type (3) discussed above. In CSP-R$_p$ processes communicate via unidirectional channels, and a channel connects exactly two processes.

D, W – stand for channel variables;
x, u – stand for program variables;
e – stands for expressions;
b – stands for boolean expressions;
p – stands for priority expressions.

Program: $P ::= S \mid N$
Statement: $S ::= x := e \mid \text{skip} \mid g \mid \text{wait} d \mid S_1 ; S_2 \mid A \mid {}^*A \mid [N]$
Guard: $g ::= b \mid D?x \mid W!e \mid \text{wait} d \mid b; D?x \textbf{ by p} \mid b;\ W!e \textbf{ by p} \mid b; \text{wait } d$
Alternative: $A ::= [\,[]_{i=1}^{n} g_i \rightarrow S_i]$
Network: $N ::= S_1 \parallel S_2$

The informal semantics of the language follows on the lines of the semantics of CSP-R given in Koymans *et al* (1988) and Huizing *et al* (1987). It may be observed that in our language, priority can be assigned to only I/O guards and not for pure boolean or delay guards since the local priorities can be manipulated through the boolean expressions. Further, we assume that pure boolean guards have priority over the I/O guards. Again, note that the priority for local action/communication can always be manipulated through the boolean parts of the expressions. For ease of understanding, we provide an informal interpretation of those commands that are quite different from those of CSP described in Hoare (1978).

Interpretation of *alternative command*: if none of the guards is open, then execution aborts; otherwise, check whether there is at least one open pure boolean guard and select one of them nondeterministically. In case there is no open boolean guard but there is at least one other open guard the execution proceeds in the following way: Compute the *waitvalue* which is defined to be infinite if there are no open wait guards; otherwise, it is defined to be the maximum of 1 and the minimum of the values of the durations of the open wait guards. Let us denote the waitvalue by d. As soon as there is a possible semantic match for the open I/O command, the communication action takes place over the channel that has the highest priority (the selection of the semantically matching commands of equal priority is nondeterministic). However, if no semantic match takes place within d units, then one of the open wait guards with waitvalue equal to d is selected nondeterministically. It may be observed that we have assumed a priority[1] for the boolean guards over the I/O guards. Note that the I/O guards can be assigned any positive priority; we assume a default priority of 1 in case there is no explicit specification. No explicit priority can be assigned to delay guards; in other words, the delay guards are assumed to have priority 0. This is consistent with the semantics of Ada.

For example, consider a system consisting of three robots: P_1, P_2, and P_3. Robots P_1 and P_2 compete for some resource service from P_3 such that P_1 or P_2 have to wait only linearly (i.e. one can wait till the other gets the service) for service. The controller for P_3 can be abstracted through the following control program in CSP-R$_p$:

$$P_3::i:=2;*[D?x\,\mathbf{by}\,i \rightarrow i:=1;\text{serve-}P_1\,[]\;W?x\,\mathbf{by}\,3\text{-}i \rightarrow i:=2;\text{serve-}P_2].$$

Informally, in the program the priority dynamically switches from communication over D to that of communication over channel W. It must be noted that this does not mean that the communications over D and W alternate. What this means is that if this program is placed in an environment wherein the communications over both D and W are available, then the net result will be that of alternating communications over D and W. However, in the context of environments wherein the communications over D and W are not always ready, the behaviour differs. In other words, one of the important points to be noted is that programs with priority clause in general cannot be transformed to a program that does not use any priority clause unless the environment is a priori given directly or indirectly. Note that if several requests having the same priority, arrive at the same time, then the choice is nondeterministic. To be more specific, we consider the possible set of actions in the following example.

$$\begin{aligned} P1::=\,&[D1?x\,\mathbf{by}\,1 \rightarrow S1\\ &[]D2?y\,\mathbf{by}\,2 \rightarrow S2\\ &[]D3?z\,\mathbf{by}\,3 \rightarrow S3] \end{aligned}$$

Let t_0 be the time of arrival at this select statement. Let us assume that some communication takes place over some channel at time $t_1 (t_1 \geqslant t_0)$. If we assume that there is no unnecessary waiting, then the general condition is that t_1 was the earliest time at which the communication could take place. The possible interpretations are given by:

1. Communication over $D3$ takes place at t_1; this does not require any other condition as $D3$ is the channel with the highest priority.
2. Communication over $D2$ takes place at t_1; this implies that communication over $D3$ was impossible since t_0.
3. Communication over $D1$ takes place at t_1; this implies that communications over $D2$ and $D3$ were impossible since t_0.

It must however be noted that how to determine the possibility of the communication or not is an implementation issue (i.e. implementation of the synchronous communication) and is not explicit in CSP-R$_p$. Note however, in Ada it is possible to determine such a possibility through the entry queues.

[1] One of the reasons for this assumption is that in our semantics for the sake of simplicity we have assumed that expression evaluation takes no time. This restriction can be removed very easily. In fact, no generality will be lost with such an assumption.

Hiding, $[N]$, has no effect on the execution of N but changes what can be observed about such executions. In other words, communications along channels in N are internalized and cannot be observed any more.

3. Model of real-time and execution model

The language CSP-R_p is a synchronous language. We assume that time proceeds in discrete time steps. This is consistent with the argument given in Pnueli & Harel (1988, pp. 84–98) that integer time domain will be appropriate for synchronous programming languages, since all the processes refer to the same global clock and operate only at certain points of time. Thus, our time domain is the set of natural numbers. For the sake of simplicity, we further assume that all primitive actions (such as assignment and communication) take one unit of time. Parameterization with respect to transmission time of the network and range of values for actions (Koymans *et al* 1988) are ignored for the sake of simplicity. In other words, real-time is modelled by relating the ith element of a history with the ith tick of a conceptual global clock. However, it should be noted that we do not either imply the existence of a global clock or assume the tightness of synchronization of the processor clocks.

A real-time execution model is useful only if we can make some assumptions about the progress each process is due to make. In this paper, we use the *maximal parallelism* model described in Koymans *et al* (1988) (we use MAXPAR as an abbreviation). In the MAXPAR model, at any instant of time all actions that can be started without violating synchronization constraints will be initiated. In other words, we assume the existence of a processor for every process. Such an assumption removes the need of considering resource scheduling. That is, a process is allowed to be idle only if all communication partners are unwilling to communicate and no local actions are possible at that point. Towards, the end of the paper, we discuss other aspects of real-time models.

4. Denotational semantics of CSP-R_p

Semantic domain

The domain consists of non-empty prefix-closed set of pairs: each pair consisting of a state and a finite history leading to this state. Infinite behaviours are modelled by their sets of finite approximations. In order to enforce maximal parallelism, we have to record whether the processes are suspended, and if so, on which communication the process is suspended etc. To enforce consistency of priority the semantics has to encode the priority information in a suitable manner so that the semantics remains compositional. These aspects are discussed formally in the following.

The domain is, $D_{\text{dom}} = P(\bar{S} \times H)$ where

- $\bar{S} = S \cup \{\perp\}$, S being the *set of proper states* (i.e. partial functions from Id (set of identifiers) to $\mathscr{V}$ (set of expression values), and
- H = set of sequences of records of the form: $\langle A, G\rangle$, where A is a set of *communication assumption records* referred to as the *Action set* and G provides the partial order

information with reference to the action set necessary for checking priority consistency.

The communication assumption records (CAR for short) are of the following types.

(1) *Communication records* of the form (D, v) where D is a channel name and $v \in \text{VAL}$ (domain of values). If the ith element is of the form (D, v), it can be interpreted as sending or receiving the value v over channel D at the ith tick of the conceptual global clock.
(2) *Ready records* of the form $R(A)$ where A is a subset of channel names. If the ith element of a history is of the form $R(A)$, it can be interpreted as the willingness of the process to communicate over the channels in A at the ith tick of the conceptual global clock, and the impossibility of communication since one of the partners is not able to communicate.
(3) *Internal moves* (denoted $\square$). If the ith element is $\square$, it can be interpreted as a local action at the ith tick of the conceptual global clock.

In other words, the observable actions are: (a) communication actions with the associated priority, (b) the time of the observable actions, and (c) the state of the time of termination.

In the a priori semantics, we keep information about the priority of the various actions in terms of triples $\langle H, D, L\rangle$ with the following interpretation:

- D is the channel over which communication is assumed to have taken place.
- H is the set of channels that have higher priority over D.
- L is the set of channels having priority less than or equal to that of D.

Note. Informally, $\langle H, D, L\rangle$ has the following interpretation.

- Processes are ready to communicate over the channels in $H \cup \{D\} \cup L$.
- Communication takes over D; that is, D is the channel that has a partner and there is no other channel that has a priority higher than D having a ready partner; H is the set of channels that have higher priority over that of D and D has a higher priority over those of L.

Obviously, sets H, $\{D\}$, and L are mutually disjoint.

Before describing the semantic domain and the equations, we formalise the priority triples.

DEFINITION 1

Consider the triple $\langle H, D, L\rangle$ where H and L are subsets of channel names and D is a channel name. Then, the triple $\langle H, D, L\rangle$ denotes the relation $\{(a, D) \mid a \in H\} \cup \{(D, b) \mid b \in L\}$.

Note. (1) By the underlying graph of $\langle H, D, L\rangle$, we mean a directed graph (V, E) where $V = H \cup \{D\} \cup L$ and E is the set of all directed edges (a, b) corresponding to (a, b) in the underlying relation. We refer to these graphs are *priority graphs.*
(2) Priority (or precedence) graphs are said to be inconsistent if they are not acyclic. We use $\perp$ to denote inconsistent priority graphs.

DEFINITION 2

Let G_1 and G_2 be priority graphs. Then,

$$join(G_1, G_2) = \begin{cases} \perp, & \text{if } G_1 \cup G_2, \text{ is not acyclic,} \\ G_1 \cup G_2, & \text{otherwise.} \end{cases}$$

In the following, we describe formally the domain.

Now,

$$\bar{S} \times H = \{\langle \sigma, h \rangle \mid \sigma \in \bar{S}, h \in H \text{ and } |h| < \infty\}.$$

A set $X \in$ STATE × HISTORY is said to be prefix-closed iff $\forall \langle \sigma, h \rangle \in X$, if $h' \leqslant h$ then $\langle \perp, h' \rangle \in X$.

The prefix-closure of X, denoted PFC (X), is defined as

$$X \cup \{\langle \perp, \lambda \rangle\} \cup \{\langle \perp, h' \rangle \mid \exists \sigma \exists h (\langle \sigma, h \rangle \in X \wedge h' \leqslant h)\}.$$

The domain consists of all nonempty prefix-closed elements of D_{dom}. Note that thc domain forms a complete lattice with set-inclusion ($\subseteq$); the lub is obtained by $\cup$ (set-union) and the least element is $\{\langle \perp, \lambda \rangle\}$.

The meaning function is of the form, $M[\![\text{Statement}]\!]: \bar{S} \to D_{\text{dom}}$ defining the meaning of statements from $\bar{S}$ to D_{dom}. For defining the meaning of alternative command compositionally, we define an auxiliary function $G[\![g, A]\!]$ from $\bar{S}$ to D_{dom} which gives the meaning of the guard g in the context of a set A of alternative guards. Denoting the set of alternative guards by A, we get:

$$G[\![g]\!]: A \to \bar{S} \to D_{\text{dom}}.$$

Notation. (1) Let $h = \langle A1, G1 \rangle \circ \langle A2, G2 \rangle \circ \cdots \circ \langle An, Gn \rangle$ be a finite history of length n. Then, we use $h^1 = A1 \circ A2 \circ \cdots \circ An$ to denote the projection of the history to the first component while we use $h^2 = G1 \circ G2 \circ \cdots \circ Gn$ to denote the projection of the history to the second component; the length of h is denoted by $|h|$, and the kth element of h is denoted by $h[k]$.

(2) The length of the history (trace) denotes the time taken for arriving at the point; the empty set is denoted by $\square$. Note that $R(\phi)$ also denotes $\square$ as well as $\langle \phi, v \rangle$ in our notation.

(3) For convenicnce, we use $\langle \sigma, A \rangle$ to denote, $\langle \sigma, \langle A, G \rangle \rangle$ when G is an empty graph.

(4) If B is a set of histories, then we use $\{\langle \sigma, B \rangle\}$ to denote $\{\langle \sigma, h \rangle \mid h \in B\}$.

(5) We represent singleton action sets of the form $\{\alpha\}$, where α is some communication assumption record (i.e., $\langle D, v \rangle$, $R(A)$, or $\square$), by α itself; we also omit the set symbol for ready records. For example, $\{\{\square\}\{\square, R(\{D, W\})\}\{\langle D, v \rangle\}\}$ is denoted by $\square\{\square, R(D, W)\}\langle D, v \rangle$. We omit the concatenation operator ($\circ$) whenever it is clear from the context.

(6) Whenever it is clear, we do not enclose the elements of the trace within the angular brackets.

The semantic equations for the language constructs are formally defined in appendix B; in the following, we informally discuss features of the parallel composition.

5. Parallel composition

Informal approach

The semantics is based on the real-time semantics discussed in Koymans *et al* (1988). The semantic domain consists of state-history pairs. The history is nothing but the traces of Koymans *et al* (1988) enriched with the priority information. In the following, we informally discuss how priority consistency is ensured in the parallel composition. Our semantics has two stages.

- *A priori semantics* – Here, we consider the semantics of each process in an isolated way.
- *Binding of the processes* – Here, the meaning of the program is obtained by considering the meaning of the component processes.

While composing the processes, we check for the mutual consistency of the isolated assumptions made in each of the processes. As already mentioned, the semantics of each process is in the domain of state-history pairs. For example, let us consider two state history pairs, $\langle s_1, h_1 \rangle$ and $\langle s_2, h_2 \rangle$ in processes P_1 and P_2, respectively. The binding of the states should be understood easily as processes do not share any common variables. Let us look at the merging of the histories. The two histories are said to be consistent iff:

(1) *Communication compatible* – That is, for every communication assumption over some channel, say D, in some process P_1 there is a corresponding matching communication partner in some process P_2 at the same time.
(2) *MAXPAR consistent* (*no unnecessary waiting*) – Check that there is no unnecessary waiting, that is, histories do not indicate a situation where both the processes are waiting for a communication that the other process can provide. This can be verified by checking that there is no common ready-record between any two histories at the same time.
(3) *Priority consistent* – Check that the histories are priority consistent, that is, histories do not indicate a situation wherein a lower priority request has been accepted in spite of the possibility of a higher priority request.

In the following, we informally show how priority consistency is ensured; communication compatibility as well as MAXPAR consistent are essentially the same as in Koymans *et al* (1988); the formal equations are given in appendix B.

For the understanding of priority consistency, let us consider the priority assumptions $\langle H, D, L \rangle$ and $\langle H', D', L' \rangle$ in the histories of any two processes at some time t. If the sets H, H', $\{D\}$, $\{D'\}$, L, L' are mutually disjoint then priority consistency follows trivially. Further, it must be noted that if the two CAR under consideration are neither communication compatible nor MAXPAR compatible then there is no need of a separate check for priority consistency. The basic idea for establishing priority consistency is to derive the underlying graph and check whether there are inconsistent (circular) precedences. The important aspect of the graph construction is that it is done incrementally and the existing graph is augmented only if it is not inconsistent after augmentation. In the following, we analyse the relations among these sets and show how inconsistent histories can be removed; a separate priority check is resorted to only if the inconsistency does not follow either from communication or MAXPAR incompatibility. For the sake of informal reasoning, we do a case analysis.

The important cases are:

(1) $H \cap H' \neq \phi$:
- In such a situation, clearly two processes are waiting unnecessarily, since D has a lower priority than those of the channels in $H \cap H'$ and D' has a lower priority than those of the channels in $H \cap H'$. In other words, this is not MAXPAR consistent. Thus, if we can ensure that the histories are MAXPAR consistent, then there is no need for checking again for priority consistency.

(2) $D' \in H$
- The situation corresponds to the situation of no partner for communication over D' – corresponding to communication incompability; hence, there is no need for a separate check for priority consistency.

(3) $D' \in L$
- This case again can be ruled out on the same lines as case (2).

(4) $D \in H'$
- This case again can be ruled out on the same lines as case (2).

(5) $D \in L'$
- This case again can be ruled out on the same lines as case (2).

(6) $L \cap L' \neq \phi$
- There is no need for checking priority consistency since by definition, we assume that communication does not take place over low priority channels.

(7) $H \cap L' \neq \phi \wedge D = D'$
- Consider the precedence graph shown in figure 1 (in the graph $\rightarrow$ denotes higher priority than) for this case. It can be easily seen that the priorities are assigned inversely in the two processes on at least one common channel – hence, inconsistent (the inconsistency can be seen due to the cycle in the graph).

(8) $L \cap H' \neq \phi \wedge D = D'$
- Inconsistency in this case follows from the previous case.

(9) $H \cap L' \neq \phi \wedge H' \cap L = \phi$
- From $H \cap L' \neq \phi$ it follows that the priority of D' is greater than or equal to that of the common channels of H and L'. By considering the second conjunct, we can conclude that D must have a higher priority than that of D'. In other words, there is a cycle and hence, inconsistent (see the priority graph shown in figure 2):

(10) $L \cap H' \neq \phi \wedge H \cap L'$
- The consistency can be ensured in the same way as the previous case.

In the above analysis, we have considered only the important situations; the other situations can be considered in a similar way. The exact way of keeping track of the information will be clear from the formal set of semantic equations. The equation for

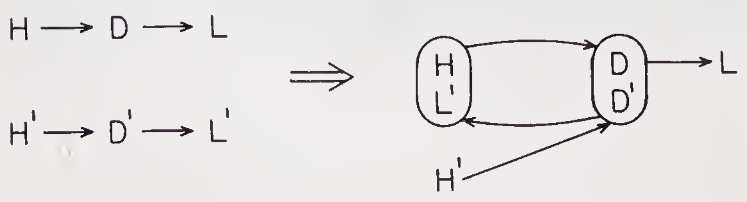


Figure 1. Priority graph for case (7).

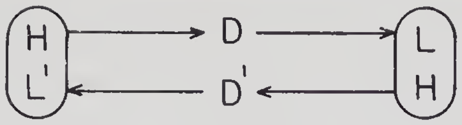


Figure 2. Priority graph for case (9).

parallel composition is given below:

$$M[\![S_1 \| S_2]\!]\sigma =$$
$$\text{PFC}(\{\langle\sigma_1\times\sigma_2, h_1^1 \# h_2^1, join(h_1^2, h_2^2)\rangle \mid \forall i\in\{1,2\}: \langle\sigma_i, h_i\rangle \in M[\![S_i]\!]\sigma$$
$$\wedge\ consistent(\sigma_1, h_1, \sigma_2, h_2)\}),$$

where

$$\sigma_1\times\sigma_2(x) = \sigma_i(x)\ \textbf{if}\ \sigma_1,\sigma_2\neq\bot \wedge x\in var(\sigma_i)$$
$$= \sigma(x)\ \textbf{if}\ \sigma_1,\sigma_2\neq\bot \wedge x\neq var(\sigma_i)\wedge x\in var(\sigma)$$
$$= \bot\ \textbf{otherwise}.$$

The pointwise merging of the histories $h_1\#h_2$ and the predicate *consistent* are defined below: Let *cset* be the set of common channels in S_1 and S_2. Then, the predicate consistent is given by,

$$consistent(h_1, h_2, cset) \underline{\Delta}\ Comm(h_1, h_2, cset)\wedge NW(h_1, h_2, cset)\wedge Pri(h_1, h_2, cset)$$

That is, h_1 and h_2 are said to be consistent iff they are communication compatible (checked by predicate *Comm*), there is no unnecessary waiting (checked by predicate *NW*) and priority consistent (checked by predicate *Pri*). Note that the consistency is checked with reference to the joint channels. Each of these predicates is defined below:

- $Comm(h_1, h_2, cset)\underline{\Delta}$

  $$\forall 1\leqslant j\leqslant max(|h_1|,|h_2|), v\in Val, D\in cset: \langle D,v\rangle\in h_1^1[j] \text{ iff } \langle D,v\rangle\in h_2^1[j].$$

  That is, for every communication on the joint channel, there is a reciprocal communication in the other process at that point of time with respect to some priority assumptions.
- $NW(h_1, h_2, cset)\underline{\Delta}\ \forall 1\leqslant j\leqslant min(|h_1|,|h_2|)$: $R(A)\in h_1^1[j]\wedge R(B)\in h_2^1[j]\rightarrow A\cap B=\phi$.
  That is, processes are not unnecessarily waiting for each other.
- $Pri(h_1, h_2, cset)\underline{\Delta}$
  $\forall 1\leqslant j\leqslant min(|h_1|,|h_2|), G_1\in h_1^2[j]\wedge G_2\in h_2^2[j]$:
  join(G_1, G_2) is an acyclic (precedence) graph.

Pointwise merging of histories: Let h_1 and h_2 be consistent with reference to the set of joint channels *cset*. Then, $h_1\#h_2$ is defined by the *j*th pointwise union as follows:

$$h_1\#h_2[j] = \langle A, G\rangle, \text{ where:}$$

(1) $A = \{\langle D,v\rangle \mid j\leqslant|h_1|\vee j\leqslant|h_2|\wedge\langle D,v\rangle\in h_1^1[j]\vee\langle D,v\rangle\in h_2^1[j]\}$.

- If the communication is over a channel not belonging to *cset* (the length of the two traces may not be equal), then the record is kept without any change; otherwise, both traces must contain the same communication record (D, v) at the time point *j* (as required by *Comm*).

$$\cup\{R(\{D \mid j\leqslant|h_1|\vee j\leqslant|h_2|\wedge(R(D)\in h_1^1[j]\vee R(D)\in h_2^1[j])\})\}.$$

- Obtain the new ready set of channels by taking the union of all the channels over which the processes are waiting; note that there is no need to check whether the channel is in *cset* or not as the consistency test has already assured (by NW) that there is not unnecessary waiting; internalizing the channel in *cset* is handled by the hiding rule.

(2) $G = join(h_1^2, h_2^2)$.

Before defining the equation for hiding, we define $hide\colon graphs \times channels \rightarrow \{\perp\} \cup graphs$.

DEFINITION 3

Let G_1 and G_2 be any two priority graphs that are consistent (i.e., precedence) and *cset* be some non-empty finite set of channels. Then, we define,

$$join_{cset}(G_1, G_2) = hide(join(G_1, G_2), cset).$$

DEFINITION 4

Let G be some priority consistent graph and *cset* be some finite non-empty set of channels. Then,

(1) $hide(\perp, cset) = \perp$.
(2) $hide(G, \phi) = G$.
(3) $\forall D \in cset\colon hide(G, cset) = hide(G', cset - \{D\})$ where

$$G' = \{(a, b) \in G \mid D \text{ is not in } \{a, b\}\} \cup \{(a, b) \mid (a, D) \in G \wedge (D, b) \in G\}.$$

Hiding Let *cset* be the set of internal channels of S. Then,

$$M[\![[S]]\!]\sigma = \text{PFC}(\{\langle\sigma, \langle A, G\rangle\rangle \mid \exists \langle A', G'\rangle\colon \langle\sigma, \langle A', G'\rangle\rangle \in M[\![S]\!]\sigma \wedge A = A' \uparrow cset \wedge G = hide(G', cset)\})$$

where $A' \uparrow cset$ is the history obtained after removing all the communications and readies on channels *cset* from A'. Note that the empty set is represented by $\square$ and hence the time points are preserved.

In the following, we sketch proofs of theorems for establishing the soundness (priority consistency) and the compositionality of the semantics. The following theorem establishes the priority consistency.

Theorem 1. *Consider an alternative command* $[\,[\!]_{i=1}^{n} g_i \rightarrow S_i]$. *Let the process be enabled on channels* $d_1, d_2, \ldots, d_m (m \leqslant n)$ *with increasing order of priority at time* t_1. *Then, communication started over* d_j *started at* $t_2 (t_2 \geqslant t_1)$ *implies that communication over* $d_{j+1}, \ldots, d_m$ *was impossible during* t_1 *to* t_2 *and communication over* d_j *was impossible during* t_1 *to* $t_2 - 1$.

Proof. The process was enabled on channels $d_1, d_2, \ldots, d_m (m \leqslant n)$ at time t_1 implies that the traces have the structure $\alpha R_{d_1, d_2, \ldots, d_m}^{t_2 - t_1 - 1} \beta$ where α and β are some traces upto t_1 and after $t_2 - 1$ respectively. It must be noted that β would have at least one non-wait action in its first component. Thus, by the predicates NW and Pri it follows that this was the earliest possible action. Note that the condition also holds when we consider hiding. Hence the theorem.

Theorem 2. *Parallel composition is associative.*

Proof. Associativity of the parallel composition ignoring the priority information follows on the lines of the proof given for CSP-R (Koymans *et al* 1988). What we need to show is that the composition of the priority information also satisfies the property of associativity. In other words, we have to prove that $join(join(G_1, G_2), G_3) = join(G_1, join(G_2, G_3))$. This follows from the fact that the union[2] of relations is associative.

Theorem 3. *The semantics is priority consistent and compositional.*

Proof. Proof follows from theorem 1 and theorem 2.

Illustrative example. In the following, we consider a simple example that illustrates the possibility of a deadlock in a network consisting of n processes, $P_1, P_2, \ldots, P_n$, for some given priority. For the sake of brevity, we ignore the values sent and received in the example. In the example, we consider only two processes P_1, and P_2 which have $\{\beta, \gamma\}$ as the common set of channels. The interesting feature of the example is that it depicts how the two processes get into deadlock for the given assignment of priorities irrespective of the behaviour of the other processes.

Example program

```
P1::= [α? by 1 →          P2::= [ω! by 1 →
      []β? by 2 →               []γ? by 2 →
      []γ! by 3 →]              []β! by 3 →]
```

$M[\![P1]\!]\sigma = \text{PFC}(\{\langle\sigma', H_1\}\})$ where H_1 consists of

$$\{\{R(\gamma,\beta,\alpha)\}^* \circ \langle\{R(\gamma,\beta), \langle\alpha,?\rangle\}, \langle\{\gamma,\beta\},\alpha,\phi\rangle\rangle,$$
$$\{R(\gamma,\beta,\alpha)\}^* \circ \langle\{R(\gamma), \langle\beta,?\rangle\}, G3\rangle,$$
$$\{R(\gamma,\beta,\alpha)\}^* \circ \langle\langle\beta,?\rangle, G4\rangle\},$$

where $G3 = \langle\{\gamma\},\beta,\{\alpha\}\rangle$ and $G4 = \langle\phi,\gamma,\{\beta,\alpha\}\rangle$ and "?" denotes that the value part is ignored.

$M[\![P2]\!]\sigma = \text{PFC}(\{\langle\sigma', H_2\rangle\})$ where H_2 consists of,

$$\{\{R(\beta,\gamma,\omega)\}^* \circ \langle\{R(\beta,\gamma), \langle\omega,?\rangle\}, \langle\{\beta,\gamma\},\omega,\phi\rangle\rangle,$$
$$\{R(\beta,\gamma,\omega)\}^* \circ \langle\{R(\beta), \langle\gamma,?\rangle\}, G7\rangle,$$
$$\{R(\beta,\gamma,\omega)\}^* \circ \langle\langle\beta,?\rangle, G8\rangle\},$$

where $G7 = \langle\{\beta\},\gamma,\{\omega\}\rangle$ and $G8 = \langle\phi,\beta,\{\gamma,\omega\}\rangle$.

The parallel composition is given by

$$M[\![P1 \,\|\, P2]\!]\sigma = \{\langle\perp,\lambda\rangle\}.$$

Explanation. For the sake of brevity, we consider only those histories that become inconsistent due to inconsistency of priority. The common set of channels of $P1$ and

[2]Note that the operation *join* is associative.

$P2$ is given by $cset = \{\beta, \gamma\}$. Most of the histories become inconsistent due to predicates *Comm* and *NW*. The following pairs of histories get eliminated due to inconsistent priorities, that is, both $join(G3, G8)$ and $join(G4, G7)$ are $\perp$. Let us take a closer look at how these two pairs of histories become inconsistent.

$$h_3 \equiv \langle\{R(\gamma), \langle\beta, ?\rangle\}, \langle\{\gamma\}, \beta, \{\alpha\}\rangle\rangle; h_8 \equiv \langle\langle\beta, ?\rangle, \langle\phi, \beta, \{\gamma, \omega\}\rangle\rangle.$$

Obviously, in $G3$, γ has priority over β, while in $G8$, β has priority over γ. Thus, there is an inconsistent partial ordering. Now, consider the pair of histories,

$$h_4 \equiv \langle\langle\beta, ?\rangle, \langle\phi, \gamma, \{\beta, \alpha\}\rangle\rangle; h_7 \equiv \langle\{R(\beta), \langle\gamma, ?\rangle\}, \langle\{\beta\}, \gamma, \{\omega\}\rangle\rangle.$$

Again here, γ has priority over β (in $G3$), while in $G7$, β has priority over γ – that is, the ordering is inconsistent.

In other words, the two processes get deadlocked in the beginning itself despite the behaviour of other processes in the network. Thus, the two processes do not do anything.

Real-time execution models

In the previous sections, we have discussed semantic specification of priority using the maximal parallelism model. In this section, we briefly discuss the effect of various parameters including those that affect maximal parallelism.

- It easily follows that ignoring priority leads essentially to the same semantics as in Koymans *et al* (1988).
- In Koymans *et al* (1988) a spectrum of models ranging from interleaving to maximal parallelism has been given accounting for the communication media and a range in timings for actions. The semantics described here can also be augmented on the same lines to account for the various parameters. However, it may be noted that in the case of the interleaving model the semantics no longer ensures Lamport's requirement (cf. Lamport 1985) *that there should not be any unnecessary waiting in realistic priority specification*; this is in conformity with Lamport's conjecture.
- Though the principle of one processor to one logical process is quite feasible, there are many situations which force resource restrictions either due to logical design (for example, recursive processes) or due to physical constraints of space. The need of resource restrictions leads to scheduling requirements. Thus, the semantics should be able to handle interrupts of statements with higher priority. One posible solution is to combine the approaches of this paper with that of Hooman (1989).
- Another aspect that one comes across in realistic situations is that of assumptions about bus-arbitration or in general fairness issues. Perhaps one can handle some of these issues in a limited way similar to that of scheduling; a thorough investigation is needed to tackle the issue of fairness in the context of compositional semantics.

6. Discussion

In this paper, we have developed a compositional denotational semantics for prioritized real-time distributed programming languages. One of the interesting features is that it extends the compositional theory proposed in Koymans *et al* (1988) for prioritized real-time languages preserving the compositionality of the semantics.

As mentioned already, the language permits users to define situations in which an action has priority over another action without the requirement of preassigning priorities to actions for partially ordering the alphabet of actions. These features are part of the languages such as Ada designed specifically keeping the needs of real-time embedded systems.

Our approach does not have the restriction (as in Cleaveland & Hennessy 1990) that only prioritized internal moves can pre-empt unprioritized actions. Our notion of priority in the environment is based on the intuition that a low priority action can proceed only if the high priority action cannot proceed due to lack of the handshaking partner at that point of execution. In other words, if some action is possible corresponding to that environment at some point of execution then the action takes place without unnecessary waiting. It must be clear that such an approach clearly satisfies Lamport's requirement (Lamport 1985) that realistic priority specification should not involve unnecessary waiting. The condition of "no unnecessary waiting" itself provides a sort of priority for unprioritized actions[3] and thus, provides a natural model for the tasking features of Ada. In the semantic theory we have proposed there is a clear distinction between the semantic model and the execution model – this has enabled us to fully ensure that there is no unnecessary waiting. It is of interest to note that the real-time semantics proposed satisfies Lamport's conditions in a natural way. Also, our work is the first formal semantics for treating priority that is state-based – thus, having advantages over algebraic approaches for reasoning about reactive systems.

We believe that the proposed compositional theory provides a sound basis for the languages for programming reactive systems (see Liu & Shyamasundar 1989). It may be observed that we have developed the semantics by considering the priorities of events in each process in an isolated manner. Thus, it is only the partial ordering of the events that is important rather than the priority number associated with the events. We have used prefix-closed sets as our semantic domain in order to treat any general reactive system.

In our semantics, we have given priority for pure boolean guards so as to be consistent with the semantics defined in Koymans *et al* (1988). This restriction can be removed very easily by appropriately changing the a priori semantics of the alternative command. The theory can also be extended to include priorities of types (1) and (2) discussed earlier. Furthermore, the semantics can be tailored to terminating systems by considering complete traces instead of prefix-closed sets. The semantic equations can be easily extended for this case (the main difference will be that we would be using greatest fix point rather than the least fix point for iteration). It may be noted that from the real-time semantics, one can obtain a temporal logic proof system on the lines of Hooman & Widom (1988). A static analysis of CSP-R$_p$ programs on the lines of the characterization in Liu & Shyamasundar (1988, pp. 134–138, 1990) can be used for deriving tools for the specification and verification of prioritized finite state systems. Currently, we are investigating the applicability of the theory to the verification of communication protocols including complex protocols such as carrier sense protocols discussed in Pnueli & Harel (1988, pp. 84–98).

[3]This should not be misunderstood as the priority model proposed; this only shows that the MAXPAR execution model has some additional advantages in the context of priority.

Appendix A

Consider the problem of minimizing the head movement in disc access which requires a different scheduling rather than the usual FIFO discipline of Ada. We briefly discuss the solution described in Gehani (1983) using the strategy of families of entries. Let us assume that the requests for service are classified into three categories declared as

```
        type REQUEST-LEVEL is (URGENT, NORMAL, LOW).
```

Urgent requests are accepted before any other kind of requests. Normal requests are accepted only if there are no urgent requests pending. Finally requests in the low category are accepted only if there are no urgent or normal priority requests pending. Within each category requests are accepted in FIFO order.

This scheme is implemented by a task SERVICE that contains the declarations of an entry family REQUEST:

```
task SERVICE is
   entry REQUEST (REQUEST-LEVEL) (D: in out DATA);
end SERVICE;
```

Each member of REQUEST handles one request category. The body of task SERVICE is given below:

```
task SERVICE is
begin
loop
   select
         accept REQUEST (URGENT) (D: in out DATA) do
            ...processtherequest
         end REQUEST;
         or when REQUEST (URGENT)'COUNT = 0 ⇒
         – the number of tasks waiting at an entry is
         – given by the COUNT attribute
         accept REQUEST (NORMAL) (D: in out DATA) do
            ...process the request
         end REQUEST;
         or when REQUEST (URGENT)'COUNT = 0 ∧
                    REQUEST (NORMAL)'COUNT = 0 ⇒
         – the number of tasks waiting at an entry is
         – given by the COUNT attribute
         accept REQUEST (LOW) (D: in out DATA) do
            ...process the request
         end REQUEST;
         ⋮
   end select;
end loop;
end SERVICE
```

Appendix B

Before describing the semantic equations, wc will define the auxiliary functions required.

Let ϕ be a function from S to D_{dom}. Then ϕ' is the function from $\bar{S}$ to D_{dom} defined by $\phi'(\sigma) = \mathbf{if}\, \sigma \in S$ then $\phi(\sigma)$ **else** $\text{PFC}(\{\langle \sigma, \lambda \rangle\})$.

Further, ϕ^* is the function from D_{dom} to D_{dom} defined by,

$$\phi^*(X) = \{\langle \sigma', h \circ h' \rangle \mid \langle \sigma, h \rangle \in X \text{ and } \langle \sigma', h' \rangle \in \phi'(\sigma)\}.$$

The function G is defined using the following two auxiliary functions.

$$\Gamma(\{b_1; \bar{g}_1, \ldots, b_n; \bar{g}_n\}, \sigma) = \{R(D) \mid \exists i\colon W[\![b_i]\!]\sigma = tt \wedge (\bar{g}_i = D!e \vee \bar{g}_i = D?x)\},$$

$$waitvalue(b; \bar{g}, \sigma) = \begin{cases} 0, & \textbf{if } \bar{g} = \lambda \wedge W[\![b]\!]\sigma = tt, \\ max(n, 1), & \textbf{if } \bar{g} = wait\ n \wedge W[\![b]\!]\sigma = tt, \\ \infty, & \textbf{otherwise}. \end{cases}$$

A priori semantics

$M[\![S]\!]\perp = \{\langle \perp, \lambda \rangle\}$ for any S; λ is the empty sequence.

$M[\![skip]\!]\sigma = \text{PFC}(\{\langle \sigma, \{\Box\} \rangle\})$.

$M[\![x := e]\!]\sigma = \text{PFC}(\{\langle \sigma[\mathscr{V}[\![e]\!]\sigma/x], \{\Box\} \rangle\})$

- where $\mathscr{V}$ is the semantic function (assumed to be given) for evaluating the arithmetic expressions.

I/O statements

$$M[\![D?x]\!]\sigma = \text{PFC}(\{\langle \sigma[v/x], \langle R(D)^t \circ \langle D, v \rangle \rangle \rangle \mid v \in Val, t \geqslant 0\}).$$

The semantics corresponds to indefinite waiting till the communication succeeds.

$$M[\![D!e]\!]\sigma = \text{PFC}(\{\langle \sigma, \langle R(D)^t \circ \langle D, \mathscr{V}[\![e]\!]\sigma \rangle \rangle \rangle \mid t \geqslant 0\}).$$

The semantics of guards is defined in terms of an environment of boolean, I/O and wait guards. We do not give the semantics of the wait statement as it follows from the semantics of the wait guard (assuming empty environment).

$M[\![g]\!]\sigma = G[\![g, \phi]\!]\sigma$, where g is an I/O command,

$M[\![S_1; S_2]\!]\sigma = M^*[\![S_2]\!](M[\![S_1]\!]\sigma)$,

$G[\![b, A]\!]\sigma = \mathbf{if}\ W[\![b]\!]\sigma\ \mathbf{then}\ \text{PFC}\{\langle \sigma, \lambda \rangle\}\ \mathbf{else}\ \{\langle \perp, \lambda \rangle\}\ \mathbf{fi}$,

- where W is the semantic function (assumed to be given) for evaluating the boolean expressions.

$$G[\![wait\ d, A]\!]\sigma =$$
$$\text{PFC}(\{\langle \sigma, \Gamma(A, \sigma)^t \rangle \mid max\{\mathscr{V}[\![d]\!]\sigma, 1\} = minwait(A \cup \{wait\, d\}, \sigma) \underline{\Delta}\ t\}),$$

- where *minwait* (A, σ) gives the minimum of the waiting periods corresponding to elements of A.

$$G[\![D?x, A]\!]\sigma =$$
$$\text{PFC}(\{\langle \sigma[v/x], \Gamma(GRDS, \sigma)^t \circ \langle \{R(HI(D, A)), \langle D, v \rangle\},$$
$$BPG(D, GRDS) \rangle \rangle \mid v \in V, 0 \leqslant t < minwait(A, \sigma)\}),$$

- where $GRDS = A \cup \{D?x \textbf{ by p}\}$, $HI(D, A)$ returns the set of channels in A which have higher priority than channel D, $EL(D, GRDS)$ returns the set of channels in $GRDS$ which have equal or lower priority than channel D, and

$$BPG(D; GRDS) = \langle HI(D, GRDS), D, EL(D, GRDS)\rangle.$$

$$G[\![D!e, A]\!]\sigma =$$
$$\text{PFC}(\{\langle\sigma, \Gamma(GRDS, \sigma)^t \circ \langle\{R(HI(D, A)), \langle D, \mathscr{V}[\![e]\!]\sigma\rangle\}, BPG(D, GRDS)\rangle\rangle \mid 0 \leqslant t < minwait(A, \sigma)\}),$$

- where $GRDS = A \cup \{D!e \textbf{ by p}\}$. The other interpretations remain the same as in the case of input guard.

$$G[\![b; g, A]\!]\sigma = G[\![g, A]\!]^*(G[\![b, A]\!]\sigma), \text{ where } g \equiv D?x \textbf{ by p} \textit{ or } D!e \textbf{ by p} \textit{ or wait } d.$$

```
$M[\![[\,[]_{j=1}^{n} g_j \rightarrow S_j]\,]\!]\sigma =$
  **if** $\vee_{j=1}^{n} W[\![g_j^b]\!]\sigma$(where $g_j^b$ is the boolean part of $g_j$) **then**
    $\cup_{j=1}^{n} M[\![S_j]\!]^*(G[\![g_j, \{g_k \mid 1 \leqslant k \leqslant n, k \neq j\}]\!]\sigma)$
  **else** $\text{PFC}(\{\langle\perp, \lambda\rangle\})$ **fi**
```

$$M[\![*A]\!]\sigma = \mu\phi.\lambda\sigma.\textbf{if}\,\exists i{:}\; W[\![b_i]\!]\sigma) = tt \textbf{ then } \phi^*(M[\![A]\!]\sigma) \textbf{ else } \text{PFC}(\{\langle\sigma, \lambda\rangle\})\textbf{fi}.$$

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

Baeten J C M, Bergstra J A, Klop J W 1985 Syntax and defining equations for an interrupt mechanism in process algebra, Report CS-R8503, Center for Mathematics and Computer Science, Amsterdam

Cleaveland R, Hennessy M 1988 Priorities in process algebras. *Inf. Comput.* 87: 58–77

Gehani N 1983 *Ada: An advanced introduction including reference manual for the Ada Programming Language* (Englewood Cliffs, NJ: Prentice Hall)

Hoare C A R 1978 Communicating sequential processes. *Commun. ACM* 21: 666–677

Hooman J 1989 A real-time semantics for multiprogramming, manuscript, REX-Concurrency Day

Hooman J, Widom J 1988 A temporal-logic based compositional proof system for real-time message passing, TR 88–919, Cornell University

Huizing C, Gerth R, de Roever W P 1987 Full abstraction of a real-time denotational semantics for an Occam-like language. ACM *Symposium on Principles of Programming Language,* (New York: ACM Press)

Koymans R, Shyamasundar R K, Gerth R, de Roever W P, Arun Kumar S 1985 Compositional semantics for real-time distributed computing. *Proc. Logics of Programs. Lecture Notes in Computer Science, vol. 197* (Berlin: Springer-Verlag)

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

Lamport L 1985 What it means for a concurrent program to satisfy a specification: Why no one has specified priority. *ACM Symposium on Principles of Programming Languages* (New York: ACM Press)

Liu L Y, Shyamasundar R K 1988 Static analysis of real-time distributed systems. *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems. Lectures Notes in Computer Science, vol. 331* (Berlin: Springer-Verlag) pp. 373–388

Liu L Y, Shyamasundar R K 1989 An operational semantics of real-time design language RT-CDL. *Fifth International Workshop on Software Specification and Design* (New York: IEEE Press)

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)

Occam 1984 *Occam Programming Manual*, Inmos Limited (London: Prentice-Hall International)

Pitassi T, Narayana K T, Shyamasundar R K 1986 A compositional semantics for Occam, TR CS-86-19, Computer Science Department, Pennsylvania State University, Pa 16802

Pnueli A, Harel E 1988 Applications of temporal logic to the specification of real-time systems. *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 331* (Berlin: Springer-Verlag) pp. 84–98

Roscoe A W 1984 *Denotational semantics for Occam. Lecture Notes in Computer Science, vol. 197* (Berlin: Springer-Verlag)

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

Sādhanā, Vol. 17, Part 1, March 1992, pp. 95–130. © Printed in India.

A hardware implementation of pure ESTEREL

G BERRY

Ecole des Mines, Sophia-Antipolis, 06565 Valbonne, France, and
Digital Equipment, Paris Research Laboratory, 85, Av. Victor Hugo, 92
Rueil Malmaison, France

Abstract. ESTEREL is a synchronous concurrent programming language dedicated to reactive systems (controllers, protocols, man-machine interfaces etc.). ESTEREL has an efficient standard software implementation based on well-defined mathematical semantics. We present a new hardware implementation of the pure synchronization subset of the language. Each program generates a specific circuit that responds to any input in one clock cycle. When the source program satisfies some statically checkable dynamic properties, the circuit is shown to be semantically equivalent to the source program. The hardware translation has been effectively implemented on the programmable active memory PERLE0 developed by J Vuillemin and his group at Digital Equipment.

Keywords. Pure ESTEREL; synchronous programming language; reactive systems; hardware implementation; mathematical semantics.

1. Introduction

ESTEREL (Berry & Cosserat 1984; Berry *et al* 1988; Berry & Gonthier 1988; Boussinot & de Simone 1991) is a synchronous programming language devoted to *reactive systems*, that is, to systems that maintain a continuous interaction with their environment by handling hardware or software events. Its software implementation is currently used in industry and education to program software objects such as real-time controllers, communication protocols (Berry & Gonthier 1989; Murakami & Sethi 1990), man-machine interfaces (Clément & Incerpi 1989), systems drivers etc. In this paper, we present a hardware implementation of the pure synchronization subset of the language that builds a specific circuit for each program. We prove the correctness of this implementation w.r.t. the mathematical semantics of the language under some conditions to be satisfied by the source program. We describe the experiments made so far and the possible uses of the hardware implementation.

1.1 *The perfect synchrony hypothesis*

ESTEREL is an imperative concurrent language with very high-level control and event manipulation constructs. It is based on a *perfect synchrony hypothesis* (Berry & Benveniste 1991), which states that control transmission, communication, and elementary computation actions take no time, or, in othcr words, that the program is conceptually executed on an infinitely fast machine. The control structures include sequencing, testing, looping, concurrency, and a powerful exception mechanism which is fully compatible with concurrency, unlike in asynchronous concurrent programming languages (Berry 1989). The primitive communication device between concurrent statements is *instantaneous broadcasting* of signals.

The perfect synchrony hypothesis is shared by the synchronous data-flow languages LUSTRE (Caspi *et al* 1987; Halbwachs 1991) and SIGNAL (Gauthier *et al* 1987; Le Guernic *et al* 1991). It makes programming very modular and flexible, and it makes it possible to reconcile input–output determinism and concurrency. This is a great benefit over classical asynchronous languages such as OCCAM or ADA that are inherently non-deterministic, a characteristic that makes reactive programming and debugging needlessly difficult, see Bcrry (1989).

ESTEREL is rigorously defined by well-analysed mathematical semantics, given in both denotational and operational styles (Berry & Gonthier 1988; Gonthier 1988).

1.2 *ESTEREL in software*

The standard ESTEREL compiler is directly based on one of the mathematical semantics. It uses sophisticated algorithms to translate a concurrent reactive program into an equivalent efficient sequential automaton that can be implemented in any conventional language. Concurrency is compiled away during this process. The resulting automaton can be directly run by actual applications. In addition to the compiler, the ESTEREL environment includes sophisticated tools such as symbolic or graphical simulators and interfaces to automata-based program verification systems such as AUTO (Boudol *et al* 1990).

1.3 *ESTEREL in hardware*

Since many CAD systems directly support automata-like specifications, ESTEREL programs can be implementcd in hardware by first translating them into automata using the standard compiler. However, this indirect translation loses most of the source concurrent structure. This is usually a good idea in software, where run-time concurrency is in fact expensive, but not in hardware, where concurrency is free and should be used as much as possible. Furthermore, there is no simple relation between the source program size and the size of the generated automaton. In the worst case, the automaton size can be exponential in the source program size, and square factors are not rare. Again, this is much more acceptable in software than in hardware.

The direct hardware implementation we present here is conceptually much better; it is based on Gonthier's (1988) semantic analysis of ESTEREL. It transforms each program into a digital circuit that exactly reflects the source concurrency and communication structure. The circuit computes the response to any input within exactly one clock cycle, however complex the program is. The translation is purely structural (compositional) and linear in size. However, it is at present limited to the pure synchronization subset of the language, which we call PURE ESTEREL, and it works only under some restrictive conditions to be satisfied by the source program.

The translation is completely formalized and proved correct w.r.t. the mathematical semantics under the above restrictive conditions. Correctness relies on the fact that perfect synchrony does not depart very much from digital circuit synchrony: zero-time is simply replaced by one cycle.

1.4 *Actual implementation and applications*

The translation from programs to circuits has been implemented within the existing ESTEREL compiler. We have run very successful experiments using the XILINX^{TM}-based PERLE0 programmable coprocessor developed at DEC Paris Research Laboratory by Vuillemin and coworkers (Bertin *et al* 1989; Shand *et al* 1990).

We are currently investigating two kinds of applications:

- Implementing existing ESTEREL programs in hardware to match high performance constraints. For example, we have directly implemented the kernel of a fast local area network protocol that was developed in ESTEREL at INRIA (Mejia Olvera 1989).
- Programming hardware controllers in ESTEREL. The language turns out to be well-adapted to programming the control part of a circuit, which is known to be difficult and error-prone with usual techniques. We show a toy example in appendix A.

The fact that the language can be implemented either in software or in hardware is useful in two respects: one can use the software programming environment to develop, debug, and verify the programs; one can experiment various trade-offs between hardware and software without changing the source code.

1.5 *ESTEREL and LUSTRE*

The LUSTRE synchronous language has also been implemented on hardware at DEC PRL, and the implementations of ESTEREL and LUSTRE are fully compatible[1]. It has to be noted that both languages differ from most existing hardware description languages by the fact that they deal only with *behaviors* and not with hardware objects, and also by the care with which they were mathematically defined and studied. The describe circuits, LUSTRE and ESTEREL are complementary: LUSTRE is well-adapted to data path description, ESTEREL is well-adapted to control automata.

1.6 *Structure of the paper*

Section 2 presents the PURE ESTEREL language and its intuitive semantics. We give enough material for the paper to be self-contained, but not to fully understand the ESTEREL programming style, referring to Berry *et al* (1988), Berry & Gonthier (1989) and to the ESTEREL documentation for these aspects. The mathematical semantics of PURE ESTEREL is given in §3. Section 4 presents an essential part of the theory of ESTEREL, the coding of states by haltsets. This coding is the root of the hardware translation, whose principle is presented by examples in §5. The translation is then formalized in §6 and proved correct in §7. We discuss the actual implementation on PERLE0 in §8 and conclude. An appendix gives the example of a simple bus interface and briefly analyzes the adequacy of ESTEREL to program hardware controllers.

[1] A by-product of our work is a translator from PURE ESTEREL into LUSTRE.

2. PURE ESTEREL

We first present signal and events which are the basic objects manipulated by PURE ESTEREL programs. We then present the kernel language on which the semantics is defined and the full language that includes kernel-definable user-friendly statements.

2.1 *Signals and events*

PURE ESTEREL deals with *signals* $\mathsf{S}, \mathsf{S}_1, \ldots$ and with *events* $E, E_1, \ldots$ that are sets of simultaneous signals. A signal that belongs to an event is said to be *present* in that event, otherwise it is said to be *absent*.

The execution of a program associates a sequence of output events with any sequence of input events. The program repeatedly receives an *input event* E_i from its environment and reacts by building an *output event* E'_i. That E_i and E'_i are synchronous is expressed by the fact that any external observer observes a *single* event $E_i \cup E'_i$. This is in particular true of any other program placed in parallel.

The production of an output event from an input event is called a *reaction*. The flow of time being entirely defined by the sequence of reactions, we also call a reaction an *instant*. This gives sense to temporal expressions such as "instantaneously" or "immediately", which mean "at the same instant", or "from then on", which means "after the current instant included", or "in the strict future", which means "after the current instant excluded".

We assume that each input event contains a special signal tick, which is therefore present at all instants. This addition to the original language of Berry & Gonthier (1988) is now supported by the ESTEREL implementation. The tick signal is analogous to the constant 1 in circuits or the constant true in LUSTRE. When programming digital circuits, it will naturally denote clock ticks.

2.2 *Modules*

The basic PURE ESTEREL programming unit is the *module*. A module has an *interface*, which specifies its input signals I, I1,... and its output signals O, O1,..., and a *body*, which is a statement that specifies its behaviour[2]. The body can use any number of local signals for internal broadcast communication. To achieve modular programming, a module can instantiate other modules as described later on. Here is a sample module definition:

```
module M:
input I1, I2;
output O1;
statement.
```

2.3 *Kernel statements*

The primitive or *kernel* PURE ESTEREL statements are:

```
nothing
halt
```

[2]There are also input/output signals, ignored here for simplicity.

```
emit S
stat₁; stat₂
loop stat end
present S then stat₁ else stat₂ end
do stat watching S
stat₁||stat₂
trap T in stat end
exit T
signal S in stat end
```

One can use brackets '[' and ']' to group statements; by default, ';' binds tighter than '||'. Both then and else parts are optional in a present statement. If omitted, they are supposed to be nothing.

The statements are imperative and manipulate control and signals. Most of them are classical in appearance. The trap-exit mechanism is an exception mechanism fully compatible with parallelism. Traps are lexically scoped.

The local signal declaration "signal S in *stat* end" declares a lexically scoped signal S that can be used for internal broadcast communication within *stat*.

2.4 *The intuitive semantics*

The intuitive semantics deals with control transmission between statements and with signal broadcasting. A statement can *start* at some instant and remain *active* until it releases the control at some further instant, either by terminating or by exiting a trap. After termination or exit, a statement becomes inactive. A statement that terminates or exits at the same instant it starts is said to be *instantaneous*. When an active statement does not terminate and exits no trap at an instant, it is said to *halt* at that instant.

The intuitive semantics is defined by structural induction on statements:

- nothing terminates instantaneously.
- halt never terminates nor exits. It always halts.
- An "emit S" statement broadcasts the signal S and terminates instantaneously.
- When started, a sequence "$stat_1$; $stat_2$" immediately starts $stat_1$ and behaves like it. If and when $stat_1$ terminates, $stat_2$ starts immediately and determines the behaviour of the sequence from then on. If and when $stat_1$ exits a trap T, so does the whole sequence, $stat_2$ being never started in this case. Notice that $stat_2$ is also never started if $stat_1$ always halts. Notice also that "emit S1; emit S2" emits S1 and S2 simultaneously and terminates instantly.
- A loop acts as an infinite sequence. When started, "loop *stat* end" immediately starts its body *stat*. When the body terminates, it is immediately restarted. If the body exits a trap, so does the whole loop. The body of a loop is not allowed to terminate instantaneously when started.
- When a "present S then $stat_1$ else $stat_2$ end" statement starts, it immediately starts $stat_1$ if S is present in the current instant and $stat_2$ if S is absent. The present statement then behaves as the corresponding branch.
- The "do *stat* watching S" watchdog statement immediately starts its body and behaves like it until the *time guard* S occurs.
  - If *stat* terminates or exits a trap strictly before S occurs, then the watching statement instantaneously terminates or exits the same trap.

— If, in the strict future of the starting instant, S occurs while *stat* is still active, then the watching statement terminates instantaneously and kills *stat*, which is not activated in the corresponding instant.

Notice two boundary problems: the guard becomes active only at the *next* instant following the starting instant; the body is *not* activated when the time guard elapses. As we shall see below, all other possibilities can be derived by combining kernel statements, which would not be true with another choice for watching.

- When started, a parallel statement "$stat_1$ || $stat_2$" immediately starts $stat_1$ and $stat_2$ in parallel. A parallel terminates instantly if and when both $stat_1$ and $stat_2$ are terminated; they can terminate at different instants, the parallel waiting for the last one to terminate. If, at some instant, one statement exits a trap T or both statements exit the same trap T, then the parallel exits T. If both statements exit distinct traps T1 and T2 at the same instant, then the parallel only exits the *outermost* of these traps, the other one being discarded.
- The statement "trap T in *stat* end" defines a lexically scoped trap T within *stat*. When the trap statement starts, it immediately starts its body *stat* and behaves like it until termination or exit. If the body terminates, so does the trap statement. If the body exits T, then the trap statement terminates instantaneously. If the body exits an enclosing trap U, so does the trap statement (traps propagate).
- An "exit T" statement instantaneously exits the trap T.
- When started, the statement "signal S in *stat* end" immediately starts its body *stat* with a fresh signal S, overriding the one that may already exist. The statement behaves as its body from then on.

A global *coherence law* relates emission and testing:

A signal is present at an instant if and only if it is received as input by the environment or emitted by the program itself at that instant.

Remarks. Notice that an emission is transient, and that there is an asymmetry between present and absent signals. There is an emit statement to set a signal present, but no statement to set it absent: by the coherence law, this is just the default.

Notice also that a loop never terminates by itself; the only way to end it is to kill it by elapsing an enclosing time guard or by explicitly exiting an enclosing trap from within the loop or from a statement placed in parallel with the loop.

Finally, notice that exiting one branch of a parallel terminates instantaneously the corresponding trap and therefore kills the whole parallel. All parallel branches are activated at the exit instant. For example, in "emit S || exit T", the left branch emits S and terminates, the right branch exits T, so that the parallel emits S and synchronizes both branches by deciding to exit T. Therefore, being killed by an exit is less severe than being killed by an enclosing watching time guard, which does *not* activate its body when elapsed.

2.5 *Examples*

The only statement that provokes halting is halt. To take a finite but non-zero amount of time, a statement must involve halt statements guarded by watching statements. The simplest example is "do halt watching S" which waits for S and terminates: by itself, the body halt would halt forever, but the enclosing "watching S" guard kills it when S occurs, and it makes the whole statement terminate. Hence the statement is guaranteed to "last exactly one S" from the time it is started (remembering that an S present when the statement starts is not taken into account). Anticipating on the definition of derived statements, we write it as "await S".

In the above example, S can be any signal, a second as well as a centimetre, a clock tick, or generally any kind of interrupt. Therefore, each signal is seen as defining its own time unit. Nesting temporal statements bearing on different time units is the main characteristic of the ESTEREL style (Berry *et al* 1988; Berry & Gonthier 1988). Here is a program that emits repeatedly O every I until reception of a signal STOP

```
do
 loop
  await I; emit O
 end
watching STOP
```

Here O is not emitted when STOP occurs, even if I is present, since the inner loop is preempted by the external watching statement at that instant.

In most event manipulation languages, the basic primitive is await, that waits for an event to *start* a computation in sequence. On the contrary, in ESTEREL, the main primitive is watching, that waits for an event to *stop* or *preempt* a computation. It is a much more powerful primitive than await. In particular, it is easy to derive await from watching, while the converse is definitely not true.

Remember the boundary problem we mentioned when describing the watching statement. To also emit O if I is present when STOP occurs, one uses a trap:

```
trap T in
  loop await I; emit O end
||
  await STOP; exit T
end
```

This works since when one branch of a parallel exits a trap that encloses the parallel, the other branch is activated in the corresponding instant before being killed. It can perform its "last wills".

The other boundary problem concerns the starting instant. If one wants the guard to be active initially, one writes

present S else do *stat* watching S end

readily abbreviated into the derived statement

do *stat* watching immediate S

The following toy example illustrates the preemption mechanism involved in concurrent exits:

```
trap T1 in
 trap T2 in
  emit S1; exit T1
 ||
  exit T2; emit S2
 end;
 emit S3
end
```

The first parallel branch emits S1 and exits T1. The second parallel branch exits T2 but does not emit S2 since an exit statement does not terminate. The body of the parallel exits simultaneously T1 and T2; since only the outermost trap matters, T2 is discarded and T1 propagates. Hence S3 is not emitted, and the outermost trap terminates with only S1 emitted.

2.6 *Full* ESTEREL

The full language has many useful derived statements. We briefly describe the most important ones. See Berry & Gonthier (1988) for the complete list and for the exact expansion into kernel statements.

Temporal statements: A temporal statement is characterized by the fact that its expansion involves present, watching, or halt kernel statements. We have already seen the simple await statement and the immediate guard variant. Here are some other useful constructs:

- Boolean expressions on signals can appear in tests or guards, as in "present S1 and S2" or "do *stat* watching not S".
- One can count occurrences of a signal (or boolean expression) within a time guard, as in "await 3 S". Occurrence counts are not discussed in this paper but are easy to handle.
- One can add a timeout clause to be executed when a watching statement terminates by elapsing its time guard and not when the body terminates by itself:

  do *stat*$_1$ watching S timeout *stat*$_2$ end

  is just an abbreviation for:

  ```
  trap T in
    do stat₁; exit T watching S;
    stat₂
  end
  ```

- The statement "do *stat* upto S" is just "do *stat*; halt watching S". Even if the body terminates, the upto statement waits for its guard to elapse.
- Deterministic event selection has the form:

  ```
  await
    case S1 do stat₁
    case S2 do stat₂
  end
  ```

  The statement waits simultaneously for S1 and S2. If one of them occurs alone, the control is instantaneously transferred to the corresponding statement. If both signals occur at the same time, the control is transferred to S1 only. This guarantees determinism.
- There are two temporal loops:

  ```
  loop stat each S
  every S do stat end
  ```

The first loop starts *stat* at once, and kills and restarts it afresh whenever S occurs. The second loop is similar but initially waits for S to start *stat*.

- The "sustain S" statement emits S continuously. It abbreviates

```
loop emit S each tick
```

General traps: There is a general exception handling mechanism that extends basic traps:

```
trap T1, T2 in
      stat
   handle T1 do stat_1
   handle T2 do stat_2
end
```

When a trap is exited, the corresponding handler is started instantaneously. Here the traps T1 and T2 are concurrent. If they are exited simultaneously, both handlers are run in parallel.

Module instantiation: Modular programming is achieved by the run statement, which instantiates a module in place, possibly invoking signal renamings:

```
run M [signal S/I]
```

A run statement terminates if and when the copied module body does.

3. The behavioral semantics

Several mathematical semantics have been developed for ESTEREL, including a denotational semantics that precisely formalizes the intuitive temporal concepts presented §2·3, see Gonthier (1988). Here we prefer to use the *behavioral semantics* (Berry & Gonthier 1988) that defines execution reaction by reaction, using Plotkin's Structural Operational Semanties technique (SOS for short). It is shown equivalent to the denotational one in Gonthier (1988).

3.1 *Form of the rules*

The behavioral semantics defines transitions of the form $M \xrightarrow[I]{O} M'$ where M is a module, I is an input event, O is the corresponding output event, and M' is a new module that will correctly respond to the next input events. In other words, M' is the new state of M after the reaction to I. The reaction $O_1, O_2, \ldots, O_n, \ldots$ to an input sequence $I_1, I_2, \ldots, I_n, \ldots$ is then defined inductively by chaining elementary reactions:

$$M \xrightarrow[I_1]{O_1} M_1 \xrightarrow[I_2]{O_2} M_2 \cdots M_{n-1} \xrightarrow[I_n]{O_n} M_n \xrightarrow[I_{n+1}]{O_{n+1}} \cdots.$$

A behavioral transition $M \xrightarrow[I]{O} M'$ is computed using an auxiliary relation $stat \xrightarrow[E]{E',k} stat'$

defined by structural induction on statements. Here E is the *current event* in which *stat* evolves, E' is the event made of the signals emitted by *stat*, and k is an integer *termination level* that codes the way in which *stat* terminates or exits and is precisely defined below.

The current event E is made of all the signals that are present at the given instant; because of the coherence law, E must contain the set E' of emitted signals, which in turn depends on E. Hence E and E' will be computed as *fixpoints*, the fixpoint equation being located in the local signal rule below.

Let *stat* be the body of M and *stat'* be the body of M'. The relation between both transition systems is as follows:

$$M \overset{O}{\underset{I}{\rightarrow}} M' \text{ iff } \mathit{stat} \xrightarrow[I \cup O \cup \{\text{tick}\}]{O,k} \mathit{stat'} \text{ for some } k$$

(under the minor restriction that no input signal is internally emitted by *stat*, see Berry & Gonthier 1988).

Termination levels

The termination level k is 0 if *stat* terminates in the current instant, 1 if *stat* halts in the current instant, and $k+2$ if *stat* exits a trap T that is k trap levels above it, i.e. if the exit must be propagated through $k-1$ traps before reaching its trap. To handle the exit level, it is useful to first decorate the exit statements with the corresponding level, as in the following example:

```
trap T in
 exit T^2
||
 trap U in
   exit T^3
 ||
   exit U^2
 end
end
```

Here the first T exit and the U exit are labeled 2 since there is no intermediate trap statement to traverse, while the second T exit is labeled 3 since one must traverse the trap U statement to reach the trap T statement. This way of handling termination is simpler than the one used in Berry & Gonthier (1988), but equivalent to it as shown in Gonthier (1988) (see also Cousineau 1980).

3.2 *Inductive rules*

The nothing statement terminates instantaneously.

$$\text{nothing} \xrightarrow[E]{\emptyset,0} \text{nothing}$$

The halt statements halts and rewrites into itself.

$$\text{halt} \xrightarrow[E]{\emptyset,1} \text{halt}$$

An emit statement emits its signal and terminates.

$$\text{emit}\, S \xrightarrow[E]{\{S\},0} \text{nothing}$$

If the first statement of a sequence terminates, the second statement is started at once; the emitted signals are merged to form the resulting emitted event, according to perfect synchrony.

$$\frac{stat_1 \xrightarrow[E]{E'_1,0} stat'_1 \quad stat_2 \xrightarrow[E]{E'_2,k_2} stat'_2}{stat_1;\ stat_2 \xrightarrow[E]{E'_1 \cup E'_2,k_2} stat'_2}$$

If the first statement of a sequence does not terminate, that is if it halts or exits a trap, the sequence behaves just as the first statement and the second statement is kept unchanged for further reactions.

$$\frac{stat_1 \xrightarrow[E]{E'_1,k_1} stat'_1 \quad k_1 > 0}{stat_1; stat_2 \xrightarrow[E]{E'_1 k_1} stat'_1; stat_2}$$

A loop instantaneously unfolds itself once. Its body is not allowed to terminate instantaneously.

$$\frac{stat \xrightarrow[E]{E',k} stat' \quad k > 0}{\text{loop}\ stat\ \text{end} \xrightarrow[E]{E',k} stat';\ \text{loop}\ stat\ \text{end}}$$

A present statement instantaneously selects its then branch if the signal tested for is present in the current instant. Otherwise, it instantaneously selects its else branch.

$$\frac{\text{S} \in E \quad stat_1 \xrightarrow[E]{E_1,k_1} stat'_1}{\text{present S then}\ stat_1\ \text{else}\ stat_2\ \text{end} \xrightarrow[E]{E'_1,k_1} stat'_1}$$

$$\frac{\text{S} \notin E \quad stat_2 \xrightarrow[E]{E'_2,k_2} stat'_2}{\text{present S then}\ stat_1\ \text{else}\ stat_2\ \text{end} \xrightarrow[E]{E'_2,k_2} stat'_2}$$

A watching statement transfers the control to its body and rewrites itself into a present statement in order to set the time guard at next instant if the body has halted.

$$\frac{stat_1 \xrightarrow[E]{E',k} stat'}{\text{do}\ stat\ \text{watching S} \xrightarrow[E]{E',k} \text{present S else do}\ stat'\ \text{watching S end}}$$

A parallel statement starts its branches instantaneously, merges the emitted signals, and returns the *max* of the termination codes. We leave it to the reader to see that this *max* operation exactly performs the required synchronization in all termination cases.

$$\frac{\mathit{stat}_1 \xrightarrow[E]{E'_1,k_1} \mathit{stat}'_1 \quad \mathit{stat}_2 \xrightarrow[E]{E'_2,k_2} \mathit{stat}'_2}{\mathit{stat}_1 \,||\, \mathit{stat}_2 \xrightarrow[E]{E'_1 \cup E'_2,\max(k_1,k_2)} \mathit{stat}'_1 \,||\, \mathit{stat}'_2}.$$

A trap terminates if its body terminates or exits the trap, that is returns termination code 2. If the body halts, so does the trap. If the body exists an enclosing trap, then the exit is propagated by subtracting 1 from the exit level.

$$\frac{\mathit{stat} \xrightarrow[E]{E',k} \mathit{stat}' \quad k=0 \text{ or } k=2}{\text{trap T in } \mathit{stat} \text{ end} \xrightarrow[E]{E',0} \text{nothing}}$$

$$\frac{\mathit{stat} \xrightarrow[E]{E',k} \mathit{stat}' \quad (k=1 \text{ and } k'=1) \text{ or } (k>2 \text{ and } k'=k-1)}{\text{trap T in } \mathit{stat} \text{ end} \xrightarrow[E]{E',k'} \text{trap T in } \mathit{stat}' \text{ end}}.$$

An exit statement returns its exit level.

$$\text{exit } \mathrm{T}^k \xrightarrow[E]{\varnothing,k} \text{halt}$$

Finally, the local signal declaration rules wind up the events E and E' according to the coherence law given in § 2·3. Within the body, they impose that a local signal is present in E if and only if it is emitted in E'. A local signal is obviously not propagated outside its declaration.

$$\frac{\mathit{stat} \xrightarrow[E \cup \{S\}]{E' \cup \{S\},k} \mathit{stat}' \quad S \notin E'}{\text{signal S in } \mathit{stat} \text{ end} \xrightarrow[E]{E',k} \text{signal S in } \mathit{stat}' \text{ end}}$$

$$\frac{\mathit{stat} \xrightarrow[E-\{S\}]{E',k} \mathit{stat}' \quad S \notin E'}{\text{signal S in } \mathit{stat} \text{ end} \xrightarrow[E]{E',k} \text{signal S in } \mathit{stat}' \text{ end}}.$$

Remarks. The resulting statement *stat′* is unused and therefore immaterial for any rule returning $k > 1$; it is discarded by the exited trap. If a rule returns $k = 0$, then its resulting term is equivalent to nothing.

Because of the intrinsic fixpoint character of the local signal rule, our inference system does not yield a straightforward algorithm to compute a transition. Given any input I one must guess the right current event E and use the rules to check that there is a correct transition. Other semantics yield finer analysis and efficient algorithms to compute the reaction; see in particular the *computational semantics* in Berry & Gonthier (1988).

3.3 *Correct programs*

Not all ESTEREL programs make sense. We say that a module M is *locally correct* if there is only one provable transition $M \xrightarrow[I]{O} M'$ for any input event I. We say that M is *correct* if it is locally correct and if all modules obtained by all possible sequences of provable transitions are locally correct.

Correctness of ESTEREL programs is a difficult issue. It is similar to correctness of digital circuits (absence of races), although much more complex because of the power of the ESTEREL instantaneous loop construct. The ESTEREL compiler checks for reasonably general sufficient correctness conditions, see Berry & Gonthier (1988). Here, we just show two examples of (locally) incorrect programs.

The following program has no fixpoint, since S should not be emitted if present and emitted if not present. It is analogous to $X = \neg X$ in circuits.

```
signal S in
  present S else emit S end
end
```

The next program has two fixpoints, one of S1 or *S2* being present in each. It is similar to $X_1 = \neg X_2$, $X_2 = \neg X_1$ in circuits.

```
signal S1, S2 in
  present S1 else emit S2 end
||
  present S2 else emit S1 end
end
```

4. The haltset coding of states

We now present an essential concept of the theory of ESTEREL, the unambiguous coding of any state by a set of control points in the original program. Technically, control points are represented by halt positions in the kernel expansion of the module body (notice that the expansion of any derived temporal statement generates at most one halt). Since ESTEREL is concurrent, a state is given by a *set* of control positions, which we call a haltset. The haltset coding is important in two respects. First, its existence shows the rationality of ESTEREL: only finitely many statements can be generated by the rewritings of a given statement. Second, it is the direct basis of the hardware implementation, and it is also heavily used in the software implementation.

The reader might skip this section at first reading and proceed directly with the informal presentation of the hardware translation in § 5. However, an understanding of the material presented here will be necessary to see why the translation is done that way and why it indeed works.

In the sequel, we consider a fixed correct module M of expanded body *stat*. For technical reasons, we assume that the body of M never terminates, adding a trailing halt if necessary. This condition does not change the observable behaviors; of course, adding a trailing halt is done after expansion and not in modules copied by M.

Call a *derivative* of *stat* any statement *stat'* that can be reached from *stat* by some sequence of reaction $\xrightarrow[I]{O}$ provable in the behavioral semantics. So far, the derivatives are defined by a rewriting process and bear no obvious structural relation with the source term *stat*. We show that any derivative can be unambiguously coded by a ***haltset H*** of ***stat***, that is by a set of occurrences of halt statements in the kernel statement ***stat***.

Consider for example the derivatives of "await S1; await S2; halt". There are three halt statements, the two first ones being respectively generated by the first and the second await. Number them 0, 1, 2. The whole statement itself will be coded by the empty haltset ϕ. The derivative that waits for S1 is

```
present S1 else
   await S1
end;
await S2;
halt
```

Its haltset will be $\{0\}$, the index of the halt generated by the active "await S1" statement. The derivative that waits for S2 is

```
present S2 else
   await S2
end;
halt
```

Its haltset will be $\{1\}$ since the second await is active. The final derivative is halt, coded by $\{2\}$. Non-singleton haltsets will be constructed by the parallel operator, which will return the union of the haltsets of its branches.

4.1 *Haltsets*

We number all occurrences of halt in ***stat*** by distinct integers from 0 to n, $n > 0$. Then a haltset H is a subset of $[0..n]$ that satisfies the following *separation* condition: If ***stat***$_1$ and ***stat***$_2$ are the two statements of a sequence or the two branches of a present test, then H cannot contain an occurrence of halt in ***stat***$_1$ together with an occurrence of halt in ***stat***$_2$.

We decorate the behavioral semantics rules by returning a haltset H when executing a numbered term. This haltset will record the places where the term has halted. The rules take the new form $\textit{stat} \xrightarrow[E]{E',k,H} \textit{stat}'$. We always return $H = \varnothing$ when $k \neq 1$ and $H \neq \varnothing$ when $k = 1$. Adding haltsets is easy for all rules except the parallel one. Executed halt statements are put into the haltset by the rule of halt and propagated by the other rules. Since the transformation is fairly obvious, we just list a few rules and leave the other ones to be reader.

$$\text{nothing} \xrightarrow[E]{\varnothing,0,\varnothing} \text{nothing}$$

$$\text{halt}^i \xrightarrow[E]{\varnothing,1,\{i\}} \text{halt}^i$$

$$\frac{\textit{stat}_1 \xrightarrow[E]{E'_1,0,\varnothing} \textit{stat}'_1 \quad \textit{stat}_2 \xrightarrow[E]{E'_2,k_2,H_2} \textit{stat}'_2}{\textit{stat}_1 ; \textit{stat}_2 \xrightarrow[E]{E'_1 \cup E'_2,k_2,H_2} \textit{stat}'_1}$$

$$\frac{stat_1 \xrightarrow[E]{E'_1,k_1,H_1} stat'_1 \quad k_1 > 0}{stat_1 ; stat_2 \xrightarrow[E]{E'_1,k_1,H_1} stat'_1 ; stat_2}$$

$$\frac{stat \xrightarrow[E]{E',k,\varnothing} stat' \quad k=0 \text{ or } k=2}{\text{trap T in } stat \text{ end} \xrightarrow[E]{E',0,\varnothing} \text{nothing}}$$

$$\frac{stat \xrightarrow[E]{E',k,H} stat' \quad (k=1 \text{ and } k'=1) \text{ or } (k>2 \text{ and } k'=k-1)}{\text{trap T in } stat \text{ end} \xrightarrow[E]{E',k',H} \text{trap T in } stat' \text{ end}}$$

For a parallel, we return the union of the haltsets returned by the branches unless one of the branches exits a trap, in which case we return an empty haltset. We make an additional technical modification explained later on: when one branch terminates, we rewrite it into nothing.

$$\frac{\begin{array}{l} stat_1 \xrightarrow[E]{E'_1,k_1,H_1} stat'_1 \\ stat_2 \xrightarrow[E]{E'_2,k_2,H_2} stat'_2 \\ H = \begin{cases} H_1 \cup H_2 \text{ if } max(k_1,k_2) \leqslant 1 \\ \varnothing \text{ if } max(k_1,k_2) > 1 \end{cases} \\ \qquad stat''_i = \begin{cases} stat'_i \text{ if } k_i \neq 0 \\ \text{nothing if } k_i = 0 \end{cases} \end{array}}{stat_1 \,||\, stat_2 \xrightarrow[E]{E'_1 \cup E'_2, max(k_1,k_2), H} stat''_1 \,||\, stat''_2}.$$

Since a module body is supposed to always halt, its global termination code must be 1. Hence the rule always returns a well-defined haltset H for any immediate derivative. This haltset is easily seen to satisfy the separation condition.

4.2 *Recovering derivatives from haltsets*

We now recover the derivative *stat'* from *stat* and H. We proceed in two steps. First we define a labeled term $stat^H$ obtained by labeling the subterms of *stat* by either $H+$ or $H-$; a subterm is labeled $H+$ if and only if it contains at least one occurrence of halt whose number is in H. If we care about the label of $stat^H$ itself, then we write it explicitly, as in $stat^{H+}$. The labels are of course redundant with H, but they make the definitions and proofs much simpler to write.

Then we define a term $R\,(stat^H)$ by structural induction on $stat^H$. Subterms labeled by – and halt statements are left unchanged.

$$R\,(stat^{H-}) = stat$$

$$R\,(\text{halt}^{iH}) = \text{halt}^i.$$

trap and local signal declaration constructs are handled by trivial structural induction.

$$R(\text{trap T in } stat \text{ end}^H) = \text{trap T in } R(stat^H) \text{ end}$$
$$R(\text{signal S in } stat \text{ end}^H) = \text{signal S in } R(stat^H) \text{ end} \quad .$$

The only non-trivial cases are:

$$R(stat_1^{H+}; stat_2^{H-}) = R(stat_1^{H+}); stat_2$$
$$R(stat_1^{H-}; stat_2^{H+}) = R(stat_2^{H+})$$
$$R(\text{loop } stat_1^{H+} \text{ end}) = \left| \begin{array}{l} R(stat_1^{H+}); \\ \text{loop } stat_1 \text{ end} \end{array} \right.$$
$$R(\text{present S then } stat_1^{H+} \text{ else } stat_2^{H-} \text{end}) = R(stat_1^{H+})$$
$$R(\text{present S then } stat_1^{H-} \text{ else } stat_2^{H+} \text{end}) = R(stat_2^{H+})$$
$$R(\text{do } stat_1^{H+} \text{ watching S}) = \left| \begin{array}{l} \text{present S else} \\ \quad \text{do } R(stat_1^{H+}) \text{ watching S} \\ \text{end} \end{array} \right.$$
$$R(stat_1^{H+} || stat_2^{H+}) = R(stat_1^{H+}) || R(stat_2^{H+})$$
$$R(stat_1^{H+} || stat_2^{H-}) = R(stat_1^{H+}) || \text{nothing}$$
$$R(stat_1^{H-} || stat_2^{H+}) = \text{nothing} || R(stat_2^{H+})$$

Notice that these definitions make sense only when the separation condition is satisfied. Notice also why we return nothing in the semantics rules when a branch terminates: this simplifies the definition of R.

Since they exactly reproduce the (new) behavioral rules right-hand side terms, one easily shows $R(stat^H) = stat'$ as expected.

We now give the main result: the coding extends from immediate derivatives to general ones. This is not completely obvious since the R operator can duplicate halts in the loop case. The result is as follows:

Theorem. *Let stat be the body of a correct program. Let H be a haltset in stat. Then for any behavioral rewriting of the form*

$$R(stat^H) \xrightarrow[E]{E',1,H'} stat'$$

the haltset H′ contains only halts occurring in stat′ and one has $stat' = R(stat^{H'})$.

Proof. The proof is by structural induction on *stat* and by case inspection on the rule applied to the whole term $R(stat^H)$ to yield $stat'$. All cases being similar, we treat the sequence and the loop as examples. We consider a given current event E.

Let first $stat = stat_1; stat_2$. There are three cases according to the labeling generated by H.

- If $stat_2$ is labeled by $H+$, then $R(stat^H) = R(stat_2^H)$. By correctness and by the hypothesis that *stat* halts, $R(stat_2^H)$ has a unique rewriting $R(stat_2^H) \xrightarrow[E]{E',1,H'} stat'$, where H' is a nonempty haltset that only contains halts in stat_2. By induction, one has $\mathit{stat}' = R(\mathit{stat}_2^{H'+})$. Since H' is all in stat_2 and nonempty, one has $R(\mathit{stat}_2^{H'+}) = R(\mathit{stat}^{H'})$ by definition of $R(\,)$ and the result follows.

- The two other cases can be grouped into one. They correspond to a term $\mathit{stat} = R(\mathit{stat}_1^H)$; stat_2, taking H as given if $\mathit{stat} = R(\mathit{stat}_1^{H+})$; stat_2 and $H = \phi$ if *stat* itself has label $H-$. By correctness, *stat* has a unique behaviour, computed by either the first or the second sequence rule. If the first sequence rule is used, then *stat'* is generated entirely by stat_2 and the results follow as in the first case. If the second sequence rule is used, the termination code of $R(\mathit{stat}_1^H)$ is 1 since *stat* halts. By induction and by the form of the rule, one has $\mathit{stat}' = R(\mathit{stat}_1^{H'+})$; $\mathit{stat}_2 = R(\mathit{stat}^{H'})$ for some nonempty H' having all its halts in stat_1. The result follows.

Assume now $\mathit{stat} =$ loop stat_1 end. There are two subcases. If *stat* is labeled by $H-$, then $R(\mathit{stat}^{H-}) =$ loop stat_1 end. The only applicable rule is the loop rule. It asks for computing stat_1, which must halt since *stat* does. By induction and by the loop rule, one has $\mathit{stat} \xrightarrow[E]{E',1,H'} R(\mathit{stat}_1^{H'+})$; *stat* for some H'. The last term is just $R(\mathit{stat}^{H'})$ as expected. If *stat* is labeled by $H+$, then $R(\mathit{stat}^{H+}) = R(\mathit{stat}_1^{H+})$; *stat*. If the first term does not terminate, we proceed as in the first loop case. Otherwise, the loop must be unfolded once and we are back again in the first loop case.

COROLLARY.

Let stat *be a module body. Then any derivative* stat' *of* stat *is equal to* $R(\mathit{stat}^H)$ *for some haltset H, and there are only finitely many derivatives.*

Proof. By induction on the length of a rewriting sequence $\mathit{stat} \xrightarrow{*} \mathit{stat}'$, since *stat* itself is equal to $R(\mathit{stat}^{\varnothing})$ and since *stat* always returns $k = 1$. The finiteness property is obvious since there are only finitely many possible haltsets.

5. Principle of the hardware implementation

In this section, we show by examples how to translate a PURE ESTEREL program into a digital circuit that computes the reaction of the program to any input in one clock cycle. The translation is structural: the circuit logical geometry is the same as that of the original program. The translation is directly based on the haltset coding theory of §4, but we present it in such a way that it can be understood independently of this coding.

We start with a first example involving only halt and watching statements. Then we show how to handle concurrency and exceptions. Finally, we indicate how to efficiently translate the full language. The formal translation is given in §6.

5.1 *A first example*

Consider the following program:

```
module M:
input I, R;
output O;
```

```
loop
  loop
    await I; await I; emit O
  end
each R.
```

After an initialization instant in which *I* is ignored, the behavior is to emit O every two I, restarting this behavior afresh each R. Expanded into kernel statements, the body becomes:

```
loop
  do
    loop
      do
        halt
      watching I;
      do
        halt
      watching I;
      emit O;
    end
  watching R
end
```

The corresponding circuit is drawn in figure 1. It has two input pins for I and R and one output pin for O. There are four kinds of cells, called Boot, Watch, Present, and Halt. Cell output pins are primed.

The Boot and Halt cells each contain one register, assuming to initially contain value 0 and to be clocked by the global circuit's clock. The other cells are purely combinatorial. The Present cells are used for present and watching source statements, each source "watching S" statement being conceptually rewritten into "watch present S"; This slight syntactic modification simplifies the cells and makes it easy to implement boolean expressions.

The circuit contains three sorts of wires: the *selection* wires s0–s5, the *activation* wires a0–a5, and the *control* wires c0–c8. The unconnected *i* and *c′* 1 pins of Halt cells correspond to other wires unused here and described later on. Whenever two wires go to the same place, they are implicitly assumed to be combined by an or gate.

The selection and activation wires go in reverse directions and form a tree that is called the *skeleton* of the circuit. This tree is determined by the nesting of halt, watching, and || statements in the source program, following the abstract syntax revealed by the source code indentation. The leftmost Halt and Watch cells correspond to the first await, the rightmost ones to the second await.

The selection wires are used to determine which part of the circuit can be active in a given state: in our example, both await statements are in mutual exclusion, and only one of them can be active at a time. When the first await is active, the wires s2, s1, and s0 are set to 1. When the second await is active, the wires s4, s3, and s0 are set to 1. The sources of the selection wires are the Halt cell registers. The upper selection wire s0 is unconnected here, but we left it there to emphasize the structural character of the translation.

The activation and control wires bear the flow of control. The activation wires

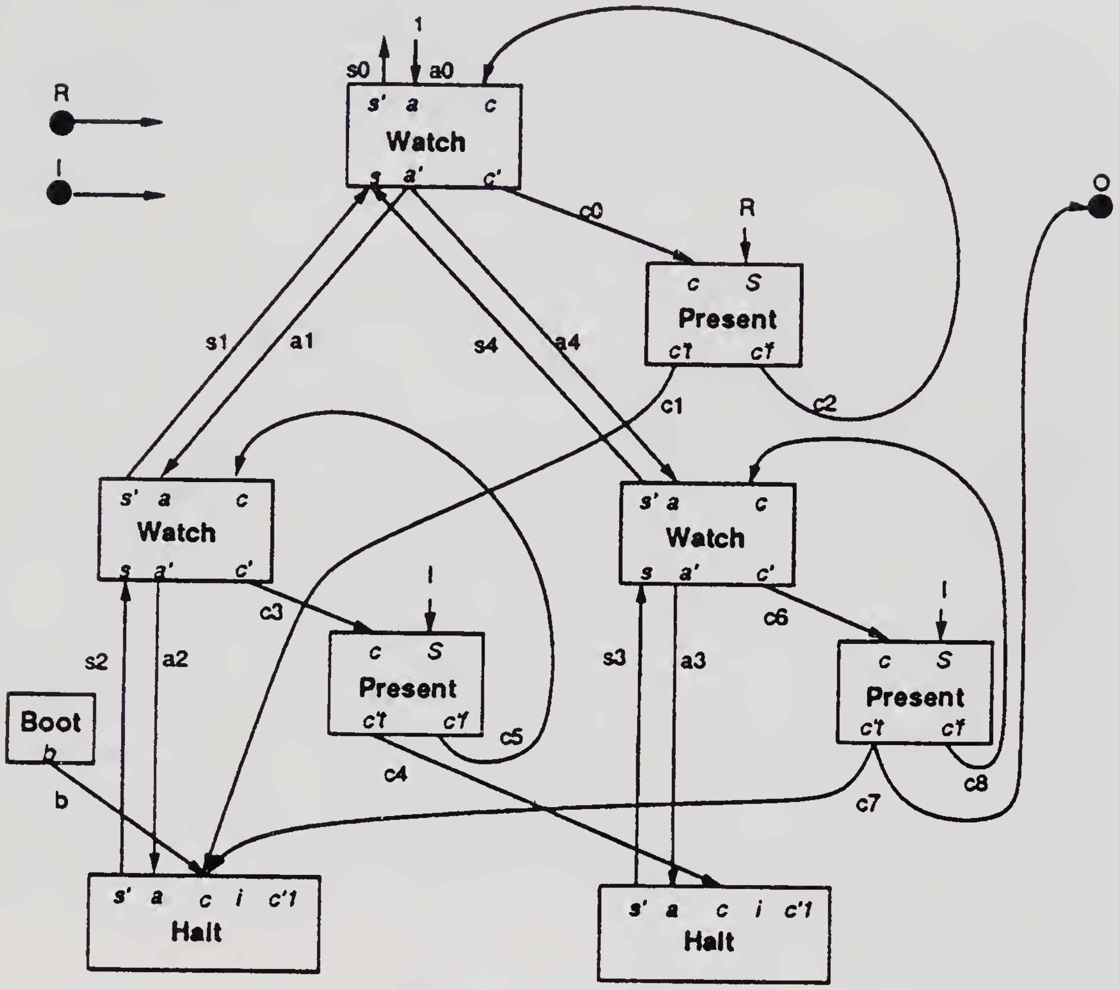


Figure 1. First example.

handle preemption between watching statements. In our example, the outermost watching preempts the innermost one; by the semantics of ESTEREL, if R is present, the outermost watching terminates without letting its body execute. The upper activation wire a0 is always set.

The cells are defined as follows:

$$\text{Boot}\begin{cases} n := 1 \\ b = \neg n \end{cases}$$

$$\text{Watch}\begin{cases} s' = s \\ c' = s * a \\ a' = c \end{cases}$$

$$\text{Present}\begin{cases} c't = c * S \\ c'f = c * \neg S \end{cases}$$

$$\text{Halt}\ \{s' := c + (a * s).$$

The notation is that of PALASM: '+' is or, '∗' is and, '¬' is not, an equality is valid at all times, and a register is denoted by ':='. Registers are supposed to contain initially 0. In the sequel, we say that a wire is *high* or *set* if it has value 1 and *low* or *reset* if it has value 0. We say that a register is *set* if it gets value 1 and *reset* if it gets value 0. Signals are assumed to be present when their wire is set and absent when their wire is reset.

The output signal b of the Boot cell is high at first clock tick and then remains low. For a Halt cell, the value of the output signal s' is initially low and then that of $c + (a * s)$ delayed one clock cycle. Hence a register is set either if an incoming control wire is set or if the activation wire is set and the register was already set[3]. The definition of *Halt* is only temporary: further pins will be added in § 5.2.

5.1a *A sample execution*: At boot time, the Halt cell registers contain 0 and the selection wires are all low; the boot control wire b is high. Because of the cell equations, all other wires are low. Hence the only effect is to set the leftmost Halt register.

On next clock tick, assume that I is present and R is absent. Then s2, s1, and s0 are set by the Halt register. Since a0 is always set, the control flows down by setting c0 that triggers the test for R in the upper Present cell. Since R is low, the control flows through the $c'f$ pin and sets c2, which is connected to the c input pin of the Watch cell. This pin is directly connected to the a' output pin, and the control flows though a1 and a4 (which are connected with each other and form in fact a single equipotential). Since both s2 and a1 are high, the leftmost Watch cell sets c3 and the leftmost Present cell sets c4 since I is present. This sets the rightmost Halt register. Since s4 is low, the rightmost Watch cell is inactive. Having no incoming control set, the leftmost Halt register is reset. This terminates the first "await I" statement.

On next clock tick, if I is present, the execution is symmetrical: the rightmost Halt is reset and the leftmost one is set. The wires set are s3, s4, a0, c0, c2, a1 = a4, c6, and c7. Since c7 is also connected to the output 0, this output is set. If instead R is present, the wires set are s3, s4, a0, c0, c1, and one is back to the state just after boot. If neither I nor R are present, then the wires set are s3, s4, a0, c0, c2, a1 = a4, c6, c8, and a3, and the state is simply restored as expected.

5.1b *Relation with the haltset coding*: Intuitively, the relation between our circuit and the haltset coding of derivatives is as follows:

- A state of the circuit is a set of Halt cells set to 1. It is therefore exactly a haltset.
- The selection wires just compute the + and − labels of statements, + being represented by a 1 in the selection wire.
- Sending the control to the translation of a subterm $stat_1$ by setting an incoming control wire amounts to execute $stat_1$. For example, setting b executes the whole statement, setting b or c1 executes the first await I, and setting c4 executes the second await I.
- Sending the control to the translation of a subterm $stat_1$ by setting its incoming activation wire amounts to execute $R(stat_1^H)$ if $stat_1$ is labeled by + in H, i.e. if the corresponding selection wire is set.

Hence, in a haltset H and an input I, the circuit just mimics the behavioral proof of $R(stat^H)$ in I. This point will be made very precise in § 7.

Notice that the Boot cell is not really necessary since the initial state can also be recognized as the only state where all Halt cells have value 0, that is where the wire s0 is low. We could as well connect the b wire to the negation of s0. However, it is convenient in practice to add the auxiliary Boot cell to reduce the length of wires and the number of logical levels.

[3]The multiplication by s is there to prevent setting the second Halt register in a term such as "do halt; halt watching S" when a is set.

5.2 *Translating parallel and exceptions*

The most complex operator is of course the parallel one, since it must synchronize the termination of its branches and propagate exceptions. Consider the following program fragment:

```
trap T in
  await S1
||
  present S2 then exit T end
end
```

The corresponding circuit fragment is shown in figure 2. The leftmost Watch-Present-Halt cell group is generated by "await S1". The rightmost Present cell is generated by "present S2", (where "else nothing" was omitted as usual). The branches are simply put in parallel and synchronized by the Parallel cell. The circuit fragment starts when it receives control by setting the c0 wire.

The Parallel cell has two parts: the fork part, which involves the six leftmost pins, and the synchronization part, which involves the eight rightmost ones.

The fork part is simple: selection wires are gathered by an or gate and activation and control are dispatched to branches.

The synchronization part is more subtle. The pins $c0$, $c1$, and $c2$ record the different termination modes according to their codes defined in section: $c0$ means termination, $c1$ means halt, and $c2$ means exiting T. With each termination pin ci is associated a

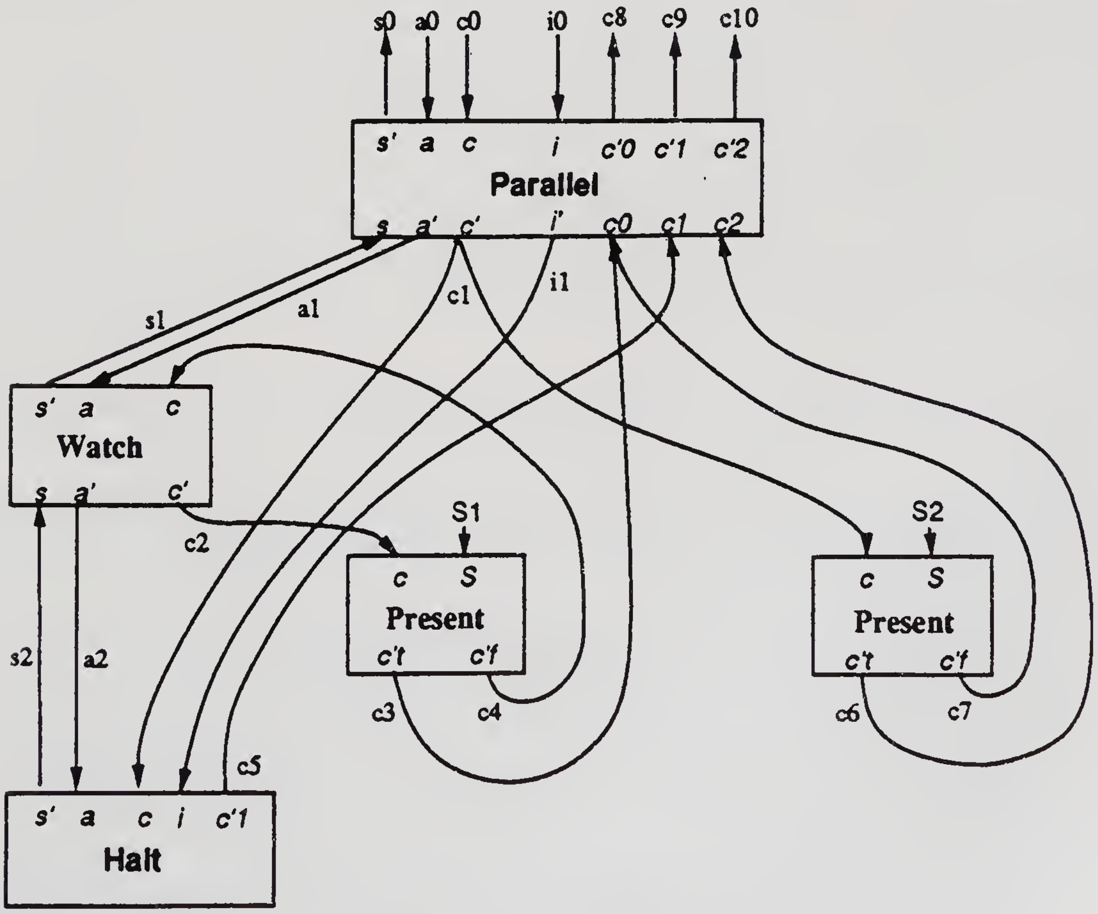


Figure 2. Second example.

continuation pin $c'i$. (In fact, $c'1$ is not really a continuation in a usual sense: it is recursively linked to the $c1$ entry of the enclosing Parallel cell when such a cell exists.)

As explained in § 3, the synchronization realized by the parallel amounts to compute the *max* of the termination codes of its branches and to only activate the corresponding continuation. It therefore uses a priority queue.

In our example, the left branch can halt, as signaled by setting wire c5, or terminate, as signaled by setting wire c3. The rightmost brach can terminate or exit T as respectively signaled by setting wires c7 and c6. Since exiting T or terminating the parallel lead to the same continuation, the continuations wires c8 and c10 will reach the same input pin in any global circuit in which our fragment is placed.

When the right branch exits T, the leftmost branch must be killed; technically, its halt statements must be removed from the current haltset. This is the role of the *inhibition* wire i1 that sends an inhibition signal to the halt register. In an actual execution context, the inhibition signal can also come from an enclosing parallel statement itself killed by some trap exit. It is then received on pin i by the wire i0.

The final equations of the Parallel and Halt cells are:

$$\text{Parallel}\begin{cases} s' = s \\ a' = a \\ c' = c \\ c'2 = c2 \\ p1 = c'2 \\ c'1 = c1 * \neg p1 \\ p0 = c1 + p1 \\ c'0 = c0 * \neg p0 \\ i' = i + p1 \end{cases}$$

$$\text{Halt}\begin{cases} c'1 = c + (a * s) \\ s' := (c + (a * s)) * \neg i \end{cases}$$

where p0 and p1 are local wires used to compute the parallel continuation and inhibition values: if ci is the selected continuation, ci is set and all continuations cj are reset for $j \leqslant i$, and i' is set if p1 is.

5.2a *A sample execution*: Assume the circuit receives control by c0 and therefore sets c1.

- Assume S2 is present. Then c5 is set by the Halt cell and c6 is set by the right Present cell. The parallel cell selects the appropriate continuation c10 and inhibits the halt register by setting i1.
- Assume instead S2 is absent. Then c5 is set by the Halt cell and c7 is set by the right Present cell. The selected continuation is c9; it signals halting to an eventual enclosing parallel statement. Since the inhibition wire i1 is low, the Halt cell register is set. The circuit then remains in the same state in further clock cycles as long as the activation wire a0 remains high and S1 remains low: the wires set are s2, s1, s0, a1, c2, c4, a2, c5, and c9. If a0 remains high and S1 is reset, the wires set are s2, s1, s0, a1, c2, c3, and c8. The whole construct terminates and the register is reset since c1 and a2 are low. The incoming activation wire a0 can also become low before S1 occurs, for example because an enclosing watchdog elapses. Then the Halt register is also reset.

5.2b *General parallel cells*: In fact, the size of the priority queue in a parallel cell depends on the number of nested traps exited from within its source parallel statement. The number of pins *ci*, *c′i* for $i \geqslant 2$ correspond to the number of cnclosing traps. With no trap, there is no such pin. The example explained one level of trap. With two levels of traps, as in

```
trap U in
  trap T in
    ···||···
  end
end
```

there would be a pin *c*2 for T and a pin *c*3 for U, and so on.

5.3 *Summary of the translation*

The translation is done by connecting together cells corresponding to source statements. The cells are the same for all programs, but the parallel cells have variable continuation parity according to the number of enclosing traps.

The logical *skeleton* of the translation is given by the tree of Halt, Watch, and Parallel cells which mimics the tree of source halt, watching, and || statements. Each edge of the tree is composed of an upward *selection* wire and a downward *activation* wire. Two sets of wires reinforce the skeleton: *control* wires that signal halting and go upwards from Halt and Parallel cells to Parallel cells, and opposite *inhibition* wires that force resetting the Halt registers in case of exceptions.

In addition to the above cells, one finds a Boot cell used to boot the circuit, and Present cells generated by source present and watching statements. These cells are linked together and to skeleton cells by *control* wires. Each Present cells also receives as input a *signal* wire. Signal wires come either from input signal pins or from local signal cells, which are simply or gates. Control wires transfer the control from cell to cell. They also emit signals by being connected to output signal pins or to local signal or gates. The wiring of control wires is determined by a continuation analysis, see § 6.

5.4 *Optimization*

The reader may find that our circuits contain lots of wires and of logical levels, even for simple programs. In fact, this is because they are obtained by a structural translation process and there is much room for automatic optimization. Many wires are simply connected with each other. Many generated logical functions are readily grouped by logic optimizers. Constant folding can also be used: for instance, the top activation wire is always set; using this fact, one can statically simplify many gates.

Therefore, our circuits should not be directly implemented; they should instead be given as input to logic optimizers. We presently use optimizers based on Binary Decision Dags (or BDD), see Brayton *et al* (1990), Coudert & Madre (1990) and Savoj *et al* (1991). They drastically reduce the actual size of circuits. They can also discover redundancies between registers and suppress some of them (Berthet *et al* 1990).

Altogether, we believe that we can obtain final circuits that are as good as carefully hand-designed ones. Because of the power and efficiency of BDD-based optimization techniques, we think there is no need to search for a more sophisticated translation process.

5.5 *The translation is sometimes incorrect*

Our translation does not translate correctly all programs. There are difficulties with local signals and with loops over parallel statements.

First, we have allocated a single wire for a local signal. But even within a single reaction, an ESTEREL signal can have several independent avatars. Consider a statement of the form

```
loop
   signal S in stat end
end
```

When the body terminates, it is restarted at the same instant with a *fresh* signal S. This is made obvious by unfolding the body to get

```
loop
   signal S in stat end;
   signal S in stat end
end
```

which is semantically equal and where there are clearly two distinct signals.

In our circuits, a signal wire has only one state at a time: we cannot implement general local signals. We must require all local signals to be declared at top level in the module body. This is not too big a restriction in practice.

The second incorrectness is more subtle. The translation of the statement

```
loop
   await S
end
```

is correct, but the translation of the equivalent statement

```
loop
   await S
||
   nothing
end
```

is not since it involves an unstable combinatorial loop through the parallel synchronizer: when S occurs, the parallel terminates and the loop makes it halt at the same timc on await S. But halting justs inhibits the termination that should provoke it, hence the combinatorial loop. Unfolding the body would solve the problem; it still builds a combinatorial loop, but this time a safe one.

The ESTEREL software checks for sufficient conditions for translation correctness. We are presently investigating a more powerful translation that will correctly translate all ESTEREL programs. It will be reported in another paper.

6. The formal translation to hardware

We define the translation formally and prove its correctness in absence of bad loops over parallels. As explained in § 5, we assume all local signal declarations to be at top level in the module body.

6.1 *Circuits*

We consider a circuit to be given by a set of *input wires*, a set of *output wires*, a set of *local wires* and a set of *wire definitions* that define output and local wires. There are two kinds of wire definitions:

- An *implication* definition $w \Leftarrow exp$ expresses a partial definition, read as "connect *exp* to *w*". There can be several implications per wire.
- A *register* definition $w := exp$ defines a wire to be initially 0 and then the value of *exp* at previous clock cycle. There can be only one register definition per wire.

Given a circuit C and a wire w, the set of implications $w \Leftarrow exp_i$ in C defines w as $w = \vee_i exp_i$. Hence the right-hand-sides of implications are connected to an or gate. If a wire w has no definition, it is considered to have an empty set of implication definitions, and is therefore to be defined by $w = 0$. To stress the fact that a wire has a single implication definition in a circuit, we can write this definition using '$=$' instead of '$\Leftarrow$'.

Given any register state and any input, the semantics of a circuit is classically defined as a unique fixpoint of the equations, and a circuit is correct if a unique fixpoint always exist in any (reachable) state. We assume this to be well-known.

6.2 *The translation environment*

The formal translation is done by natural semantics inference rules (Kahn 1988). The sequents have the form $\rho \vdash \textbf{\textit{stat}} \rightarrow C$, where ρ is a wire environment, ***stat*** is an ESTEREL statement, and C is the resulting circuit.

As in natural semantics or in PROLOG, allocation of new wires is implicit and done when encountering free variables. To make things clear, we shall comment on each rule and explicitly tell which are the newly allocated wires.

The environment ρ is made of several wires, whose functions have been explained in § 5. It contains the following fields

- An incoming control wire c.
- A selection wire s.
- An activation wire a.
- An inhibition wire i.
- A vector of continuation wires $\mathbf{c}$. The wire $\mathbf{c}^0$ corresponds to termination, the wire $\mathbf{c}^1$ corresponds to halting, the wire $\mathbf{c}^{k+2}$ corresponds to exit $k+2$, that is to exiting k trap levels.
- A set of signal wires S, one for each input, output, or local signal S. For simplicity, we assume that all local signals have distinct names; then all local signal wires can be preallocated.

We use the classical dot notation to get environment components: for instance, $\rho.c$ denotes the control wire of ρ. Given an environment ρ, we shall often need to consider another environment ρ' that differs from ρ by the value of one field, say by changing $\rho.c$ into c'. We then write $\rho' = \rho[c \leftarrow c']$. The notation extends naturally when changing several fields.

To translate a module, we allocate a boot control wire b and a register n of equations $\mathsf{b} = \neg\, \mathsf{n}, \mathsf{n} := 1$ as in § 5, a dummy selection wire s, two dummy wires $\mathsf{c0}$ and $\mathsf{c1}$ for the (unused) continuations, a dummy inhibition wire i, and one wire S per signal, declaring respectively input and output signals as inputs and outputs to the circuit. We translate the module body in the environment

$$\rho_0 = (\mathsf{b}, \mathsf{s}, 1, \mathsf{i}, (\mathsf{c0}, \mathsf{c1}), \mathbf{S}).$$

6.3 *The translation rules*

The cells of § 5 were useful for an intuitive explanation, but in rules it is simpler to produce equations directly.

For a nothing statement, we connect the incoming control to the termination continuation wire.

$$\rho \vdash \mathsf{nothing} \rightarrow \rho.\mathbf{c}^0 \Leftarrow \rho.c$$

For a halt statement, we connect the incoming control to the halt continuation wire, to signal halting to an enclosing parallel statement. We allocate a new selection wire s' defined as a register with input as explained in § 5. We connect it to the environment selection wire $\rho.s$.

$$\rho \vdash \mathsf{halt} \rightarrow \left| \begin{array}{l} \rho.\mathbf{c}^1 \Leftarrow \rho.c + (\rho.a * \rho.s) \\ \rho.s \Leftarrow s' \\ s' := (\rho.c + (\rho.a * \rho.s)) * \neg\, \rho.i. \end{array} \right.$$

For an emit statement, we connect the incoming control to the termination wire and to the signal wire.

$$\rho \vdash \mathsf{emit}\ \mathsf{S} \rightarrow \left| \begin{array}{l} \rho.\mathbf{c}^0 \Leftarrow \rho.c \\ \rho.S \Leftarrow \rho.c \end{array} \right. .$$

For a sequence, we allocate a new wire c' for control transmission. We translate the first statement with c' as termination and the second statement with c' as incoming control.

$$\frac{\begin{array}{l} \rho[\mathbf{c}^0 \leftarrow c'] \vdash \mathit{stat}_1 \rightarrow C_1 \\ \rho[c \leftarrow c'] \vdash \mathit{stat}_2 \rightarrow C_2 \end{array}}{\rho \vdash \mathit{stat}_1 ; \mathit{stat}_2 \rightarrow \left| \begin{array}{l} C_1 \\ C_2 \end{array} \right.}$$

For a loop, we allocate a new wire c' to handle looping and we connect the incoming control to it. We translate the body with c' both as incoming control and as outgoing continuation.

$$\frac{\rho[c \leftarrow c', \mathbf{c}^0 \leftarrow c'] \vdash \mathit{stat} \rightarrow C}{\rho \vdash \mathsf{loop}\ \mathit{stat}\ \mathsf{end} \rightarrow \left| \begin{array}{l} c' \Leftarrow \rho.c \\ C \end{array} \right.}$$

For a present statement, we allocate two new control wires c_1 and c_2; then c_1 is set when the incoming control is present and the signal is present, while c_2 is set when the incoming control is present and the signal is absent. We translate the branches with c_1

and c_2 as respective incoming controls.

$$\frac{\begin{array}{c}\rho[c\leftarrow c_1]\vdash \mathit{stat}_1\rightarrow C_1\\ \rho[c\leftarrow c_2]\vdash \mathit{stat}_2\rightarrow C_2\end{array}}{\rho\vdash \text{present S then } \mathit{stat}_1 \text{ else } \mathit{stat}_2 \text{ end}\rightarrow \left|\begin{array}{l} c_1=\rho.c*\rho.S\\ c_2=\rho.c*\neg\rho.S\\ C_1\\ C_2\end{array}\right.}.$$

For a watching statement, we allocate a new selection wire s' and connect it to $\rho.s$, and we allocate a new activation wire a'. The outgoing activation wire a' is set if s' and $\rho.a$ are set and the signal is absent. The outgoing termination wire $\rho.\mathbf{c}^0$ is set if s' and $\rho.a$ are set and the signal is present.

$$\frac{\rho[s\leftarrow s', a\leftarrow a']\vdash \mathit{stat}\rightarrow C}{\rho\vdash \text{do } \mathit{stat} \text{ watching S}\rightarrow \left|\begin{array}{l} \rho.s\Leftarrow s'\\ \quad a'=\rho.a*\rho.s*\neg\rho.S\\ \rho.\mathbf{c}^0\Leftarrow\rho.a*\rho.s*\rho.S\\ C\end{array}\right.}.$$

The parallel rule is of course the most complex one. It follows exactly the intuitive explanation given in § 5. We allocate a selection wire s' connected to $\rho.s$, an inhibition wire i', a continuation vector $\mathbf{c}'$ of the same length k as $\rho.\mathbf{c}$, and a priority vector $\mathbf{p}$ of length $k-1$. We recursively translate the body with the new selection, inhibition, and continuation wires. Then we establish the priority queue to compute the outgoing continuations and the new inhibition wire i'.

$$\frac{\begin{array}{l} \qquad k=|\rho.\mathbf{c}|\\ \rho[s\leftarrow s', i\leftarrow i', \mathbf{c}\leftarrow\mathbf{c}']\vdash \mathit{stat}_1\rightarrow C_1\\ \rho[s\leftarrow s', i\leftarrow i', \mathbf{c}\leftarrow\mathbf{c}']\vdash \mathit{stat}_2\rightarrow C_2\end{array}}{\rho\vdash \mathit{stat}_1 \| \mathit{stat}_2\rightarrow \left|\begin{array}{l} \quad\rho.s\Leftarrow s'\\ \rho.\mathbf{c}^{k-1}\Leftarrow\mathbf{c}'^{k-1}\\ \quad\mathbf{p}^{k-2}=\mathbf{c}'^{k-1}\\ \rho.\mathbf{c}^{k-2}\Leftarrow\mathbf{c}'^{k-2}*\neg\mathbf{p}^{k-2}\\ \quad\mathbf{p}^{k-3}=\mathbf{c}'^{k-2}+\mathbf{p}^{k-2}\\ \qquad\qquad\ldots\\ \quad\mathbf{p}^{0}=\mathbf{c}'^{1}+\mathbf{p}^{1}\\ \quad\rho.\mathbf{c}^{0}\Leftarrow\mathbf{c}'^{0}*\neg\mathbf{p}^{0}\\ i'=\begin{cases}\rho.i & \text{if } k\leqslant 3\\ \rho.i+\mathbf{p}^1 & \text{if } k>3\end{cases}\\ C_1\\ C_2\end{array}\right.}.$$

For a trap, we shift by 1 all wires in $\rho.\mathbf{c}$ after position 2 and we insert the termination continuation $\rho.\mathbf{c}^0$ at exit position 2. The vector notations are obvious.

$$\frac{\rho[\mathbf{c} \leftarrow (\rho.\mathbf{c}^0, \rho.\mathbf{c}^1, \rho.\mathbf{c}^0)\cdot\rho.\mathbf{c}^{2..}] \vdash \mathit{stat} \rightarrow C}{\rho \vdash \mathsf{trap\ T\ in}\ \mathit{stat}\ \mathsf{end} \rightarrow C}.$$

For an exit, we connect the incoming control to the appropriate continuation.

$$\rho \vdash \mathsf{exit\ T}^k \rightarrow \rho.\mathbf{c}^k \Leftarrow \rho.c.$$

For a local signal declaration, we simply translate the body since the signals have been pre-allocated.

$$\frac{\rho \vdash \mathit{stat} \rightarrow C}{\rho \vdash \mathsf{signal\ S\ in}\ \mathit{stat}\ \mathsf{end} \rightarrow C}.$$

7. Correctness of the translation

We first explain roughly the proof idea as if the translation was always correct. Consider the body *stat* of a correct module placed in the initial environment where the local signal wires have been cut. Then there are two separate wires for each local signal, one for input and one for output. Consider a signal environment E and a haltset H. There exists a unique behavior $\mathit{stat} \xrightarrow[E]{E',1,H'} \mathit{stat}'$ with $\mathit{stat}' = R(\mathit{stat}^{H'})$, and a unique behavior $R(\mathit{stat}^H) \xrightarrow[E]{E',1,H''} \mathit{stat}''$ with $\mathit{stat}'' = R(\mathit{stat}^{H''})$; uniqueness is obvious since there are no local signal declarations in stat_1.

The circuit fragment $C(\mathit{stat})$ obtained by translating stat_1 has two incoming control wires c and a. Then setting c realizes the first behavior, while setting the activation wire a realizes the second behavior. Furthermore, because of loops, c and a can be both set. Then the circuit sums up both behaviors with no interference between them. The proof goes simply by structural induction.

Once this is shown, close the local signal wires. Then, for the module body *stat*, for any state H and real input event I, there exists a unique local event L and a unique output event O such that

$$R(\mathit{stat}^H) \xrightarrow[I \cup L \cup O \cup \{\mathrm{tick}\}]{O,1,H'} R(\mathit{stat}^{H'})$$

But closing the local signal wires in the circuit has exactly the same coherence effect as in the semantics: a signal is there if and only if it is emitted. Since the circuit can do nothing but mimic the behavioral semantics and since there is only one fixpoint in the semantics by the correctness hypothesis, there is only one fixpoint in the circuit and it is the required one[4].

Therefore, one can view the circuit as a *folding of all possible behavioral semantics proof trees* of a program and of its residuals in all possible environments. What the electrons do is to select the right proof tree in one clock cycle given a residual and an input.

[4] We talk here of abstract circuits, or equivalently we assume that concrete circuits always find the unique fixpoint when it exists.

The only problem with the above proof argument is that sending control to a parallel by both c and a does *not* sum up the behaviors: one of the continuations can be discarded by the other one. Here, we shall simply prove that the circuit works fine under the assumption that the problem can never appear dynamically[5]. This leads to the following condition:

Condition NSP. A correct program is said to be NSP (Non Schizophrenic for Parallels) if for any haltset H and for any event E, no parallel subterm $\mathit{stat} = \mathit{stat}_1 \,\|\, \mathit{stat}_2$ that contains a halt in H is evaluated in the behavioral semantics proof of the reaction of the module body under E both under the form stat and under the form $R(\mathit{stat}^{H+})$.

This is certainly a strange and non-structural condition, but its main advantage is to be amazingly trivial to check in the ESTEREL software compiling process. We have put an appropriate specific option in the ESTEREL compiler to report its failure.

Theorem. *For any correct* NSP ESTEREL *module M, the circuit C(M) has exactly the same input-output behavior as M.*

Proof. The proof goes just as sketched, but we must inductively ensure that no parallel receives c and a together.

We first study the circuit reactions when the local signal wires are opened. We consider a given haltset H and a given input event E. Let P be the proof of $R(\mathit{stat}^H) \xrightarrow[E]{E',1,H'} R(\mathit{stat}^{H'})$.

Given a subterm stat_1 of stat, define the type of stat_1 in P as follows: stat_1 is of type *null* if it does not appear in P, of type c if it appears in P only under the form stat_1, of type a if it appears in P only in the form $R(\mathit{stat}^{H+})$, and of type ca if it appears in both forms.

For the circuit $C(\mathit{stat}_1)$ generated by stat_1, we say that we send the control *null* if we set neither c nor a, the control c if we set c and not a, the control a if we set a and not c while s is set, and the control ca if we set both c and a while s is set.

We show the following properties on any subterm stat_1, by structural induction:

(a) If stat_1 receives the control as indicated by its type in P, then it will itself send the control to all its subterms as indicated by their type in P.
(b) Under control *null*, $C(\mathit{stat}_1)$ sets no continuation, no signal, and no halt.
(c) If stat_1 is of type c and $\mathit{stat}_1 \xrightarrow[E]{E'_c,k_c,H_c} \mathit{stat}'_1$, then, under control c, $C(\mathit{stat}_1)$ emits E'_a, sets the sole continuation $\mathbf{c}^{k_a}$, and sets exactly the halts in H_c iff its incoming inhibition wire i has value 0.
(d) If stat_1 is of type a and $R(\mathit{stat}_1^{H+}) \xrightarrow[E]{E'_a,k_a,H_a} \mathit{stat}'_1$, then, under control a, $C(\mathit{stat}_1)$ emits E'_a, sets the sole continuation $\mathbf{c}^{k_a}$, and sets exactly the halts in H_a iff its incoming inhibition wire i has value 0.
(e) If stat_1 is of type ca, then, under control ca, $C(\mathit{stat}_1)$ realizes the union of the behaviors of case (c) and (d).

[5]The right solution would be to use *two* synchronizers, one for c and one for a, and to duplicate some of the logic of the body to signal termination to the appropriate synchronizer; in fact, one must use more than two synchronizers in the general case to properly handle parallel statement nesting; this will be the subject of a forthcoming paper.

First notice some general facts. The s wire is set for $stat_1$ iff $stat_1^{H+}$. Hence only statements that contain halts in H will receive both a and s. By construction, any circuit $C(stat_1)$ does nothing under control *null* and sets no halt when its incoming inhibition wire i is set; otherwise, its sets its halts normally. Also, since all statements merge their emitted signals by or gates, the signal behavior will always be the expected one.

The statements nothing, emit S, and exit T are always of type *null* or c and they exhibit the (c) behavior under c. A halt can be of any type, but it always sets $\mathbf{c}^1$ and its register if $i=0$ as required under control c, a, or ca.

Consider a sequence $stat_1$; $stat_2$ of type c. Then $stat_1$ is itself of type c, and the induction tells that $C(stat_1)$ behaves just as $stat_1$ under c. If $stat_1$ terminates, then $stat_2$ is of type c since the first sequence rule must be applied in the proof (it cannot be of type ca, otherwise the sequence itself would be of that type). But $C(stat_1)$ sets $\mathbf{c}^0$ that starts $stat_2$ under control c by the sequence wiring. The induction shows (c). If $stat_1$ does not terminate, then $stat_2$ is of type *null* and $C(stat_2)$ receives no control and does nothing; hence the sequence behaves just as $stat_1$, which shows (c). Condition (a) also follows from this case analysis.

The proof of (d) and (a) is similar for a sequence of type a, analysing separately the cases $stat_1^{H+}$ and $stat_2^{H+}$.

Consider finally a sequence of type ca. First assume $stat_1^{H+}$. Then $stat_1$ itself is of type ca, and the induction applies to it. Furthermore, $stat_2$ is started under c iff $stat_1$ terminates under c, a, or both. But giving twice the control to $stat_2$ is just the same as giving it once, since incoming control wires are gathered by an or gate, and (e) follows. Next assume $stat_2^{H+}$. Then $stat_1$ is of type c, while $stat_2$ is of type a if $stat_1$ does not terminate, making (e) obvious, and of type ca if $stat_1$ terminates; in the latter case, (e) is established by induction on $stat_2$. The case analysis is finished for the sequence, and it also shows (a) in all cases.

The other operators are handled in the same way. For a parallel, one is never in case (d) by the NSP hypothesis, and one remembers that the i wire is set in case of exit to kill the haltsets of the subterms.

Finally, as explained before, the circuit is forced to compute the same fixpoint as the behavioral semantics when closing the local signal wires. To finish the proof, just notice that the module body *stat* receives c at the first instant from the boot wire and a at the next instants from the selection wire that is plugged back as the activation wire.

8. Implementation

8.1 *Actual implementation on* PERLE0

We have experimented our hardware implementation on the PERLE0 board developed at DEC PRL (Bertin *et al* 1989). It consists of a set of 25 synchronous XILINX programmable logic cell arrays placed on a board and piloted by a SUN^{TM} workstation.

The translation is performed by the *strldg* processor (ESTEREL to digital), which is integrated in the standard ESTEREL compiler[6]. The generated logical circuit is printed out in PERLE0 format and translated into XILINX native format by the PERLE0 software

[6] In fact, most of the skeleton and continuation analysis is already done by the standard compiler first pass.

(we could as well produce portable formats such as PALASM). The logical circuit is then given to optimizers and the optimized result is fed into an automatic placer-router, without any pre-placing indication. This gives a XILINX circuit specification. Using this environment, the turnover is of the order of 15 minutes from source program to running circuit for a medium-size program.

On PERLE0, we provide a symbolic debugging and exact speed measure environment, with interactive symbolic input and output from within Lisp or C. The speed measurc reports at which maximal clock speed a circuit correctly handles a benchmark. In practice, the speed is 30 to 75 nanoseconds for a small program (30 ns for the circuit presented in the appendix), and 75 to 100 nanoseconds for a medium size program that still fits into a single chip (about 2–4 pages of source ESTEREL code), this on a 3020 XILINX chip.

In debug or speed-measure mode, the ESTEREL program is implemented on a single chip and other chips are devoted to bus and debug interfaces. The applications we have handled so far are man-machine interfaces, real-size local area network controllers (Mejia Olvera 1989), and various circuit controllers including those used in the PERLE0 board itself to communicate with the bus and with the tested program.

8.2 *Simulation and correctness proofs*

ESTEREL and LUSTRE are themselves able to describe digital hardware. The strldg processor is also able to unparse the circuit in ESTEREL or LUSTRE. There are two main uses:

- After compiling the ESTEREL version of the circuit, we can used the full ESTEREL programming envirorment to perform simulations, analysis, and optimizations.
- Once the circuit behavior automaton is generated by ESTEREL compiler, we can use the AUTO verification system to automatically check for equivalence between the source code and the circuit automata (Boudol *et al* 1990). This may seem unnecessary since the translation has been mathematically proved correct, but software is software and double-checks are always useful. Furthermore, the translation can work properly even if the sufficient correctness conditions are not met. If AUTO reports equivalence, the circuit is perfectly usable even if it works by chance!

Of course, using the ESTEREL standard compiler for such a circuit unparsing analysis makes sense only if the circuit has a reasonable number of states, say 50 to 500, which is usually the case for controllers.

9. Conclusion

Although ESTEREL was not at all designed as a hardware description language, the work presented here shows it is well-suited to very high-level verified hardware generation. The hardware implementation is directly based on the formal semantics. The electrons circulating in the wires perform the computation of the proof tree associated with a program and an input within a single clock cycle. The circuit itself can be viewed as a folding of all possible semantical proof trees into a graph structure.

The translation we have presented is not general since programs are assumed to obey a sufficient NSP condition; we are now in the process of releasing a full correct translation of ESTEREL into circuits, based on extensions of the same ideas.

We investigate three main kinds of applications: implementing existing ESTEREL programs on hardware to improve their performance, using ESTEREL to directly program hardware controllers, and using ESTEREL to build reference controllers to which actual hand-tailored controllers can be automatically proved equivalent. Our present experiments are very promising and leave place for sophisticated optimization.

To our knowledge, the closest related works are the hardware implementation of LUSTRE and SML (Clarke *et al* 1991). The LUSTRE and ESTEREL implementations are developed in parallel and are fully compatible. Compared to SML, ESTEREL is much more elaborate as a programming language, having in particular watchdogs, exceptions, and instantaneous broadcast. Our implementation is direct and does not use a translation to automata, although such a translation is also available. LUSTRE, SML, and ESTEREL all give access to temporal logic or process calculi based verifiers. We need more experience to compare the relative qualities of the languages and of their verification tools.

This work was motivated by discussions with Jean Vuillemin and Patrice Bertin from DEC Paris Research Laboratory. It owes much to the work of Georges Gonthier on the semantics of ESTEREL. The actual implementation on PERLE0 was done at DEC PRL under the supervision of Patrice Bertin, who provided invaluable help. The experiments with BDD optimizers were conducted by Olivier Coudert and Jean-Christophe Madre (BULL CRG), as well as by Hervé Touati (DEC PRL).

Appendix A.

A1. *A simple bus interface example*

As a toy application example, we program an interface module between a bus and a hardware application. This interface is a slight simplification of the one effectively used in the PERLE0 board to run actual ESTEREL program hardware translations. Although the program is very small, we use submodules to illustrate modular programming.

A1.1 *The interface informal specification*

The interface repeatedly waits for input from the bus, tells the application to store the corresponding data word, triggers a computation, and tells the application to send back the output data word to the bus when the computation is terminated and the bus is ready for output.

The interface receives two signals from the bus, BUS_WRITE for input and BUS_READ for output. It acknowledges both input and output by sending back BUS_ACK.

Data words are received or emitted directly by the application. To control data input, the interface tells the application to connect its input buffers to the data bus by setting a signal OPEN_INPUT. This signal is maintained until the arrival of BUS_WRITE included. After one clock cycle, the interface sends BUS_ACK and starts the computation by sending a signal GO to the application. When the computation is terminated, the application sends back a signal FINISHED. The output data is then ready in the application output buffers. The interface tells the application to connect its output buffers to the bus by sending a signal OPEN_OUTPUT. This can be done only when the computation is finished and when the bus has sent BUS_READ. After waiting a clock cycle for the data to be effectively present on the bus, the interface sends BUS_ACK.

In addition, we assume that the bus can send at any time a RESET signal telling the interface to reset itself to its initial state.

A1.2 *The interface ESTEREL program*

The interface module is written as follows:

```
module Interface:
input BUS_READ, BUS_WRITE, RESET;          % from bus
output BUS_ACK;                            % to bus
output OPEN_INPUT, OPEN_OUTPUT, GO;        % to application
input FINISHED;                            % from application

loop
  loop
    run Input;
    run ComputeAndOutput
  end
each RESET.
```

Notice that the RESET signal is completely factored out and effectively resets the interface independently of its current internal state.

The Input submodule is written as follows:

```
module Input:
input BUS_WRITE;        % from bus
output BUS_ACK;         % to bus
output OPEN_INPUT;      % to application
trap INPUT in
  sustain OPEN_INPUT
||
  await BUS_WRITE do exit INPUT end
end;
await tick;
emit BUS_ACK.
```

Here we use a trap construct to ensure that OPEN_INPUT is emitted when BUS_WRITE is received. One could write as well:

```
do
  sustain OPEN_INPUT
watching BUS_WRITE;
emit OPEN_INPUT;
```

By the semantics of the watching construct, the statement "sustain OPEN_INPUT" is not executed when BUS_WRITE occurs. This is why OPEN_INPUT must be explicitly emitted at that instant.

The ComputeAndOutput module is written as follows:

```
module ComputeAndOutput:
input BUS_READ;               % from bus
output BUS_ACK;               % to bus
output GO, OPEN_OUTPUT;       % to application
input FINISHED;               % from application
[
  await BUS_READ
||
  emit GO;
  await FINISHED;
];
emit OPEN_OUTPUT;
await tick;
emit BUS_ACK.
```

Notice how the parallel statements realize the synchronization: it terminates exactly when the computation is finished and the bus is ready to read.

Once optimized, placed, and routed, the circuit uses up 9 cells on a XILINX 3020 circuit. There are 5 registers and 11 logical functions with a total of 35 inputs.

A1.3 *The advantages of* ESTEREL

The automaton generated by the ESTEREL compiler is pictured in figure A1. Notice the diamond generated by the parallel statement that appears in ComputeAndOutput.

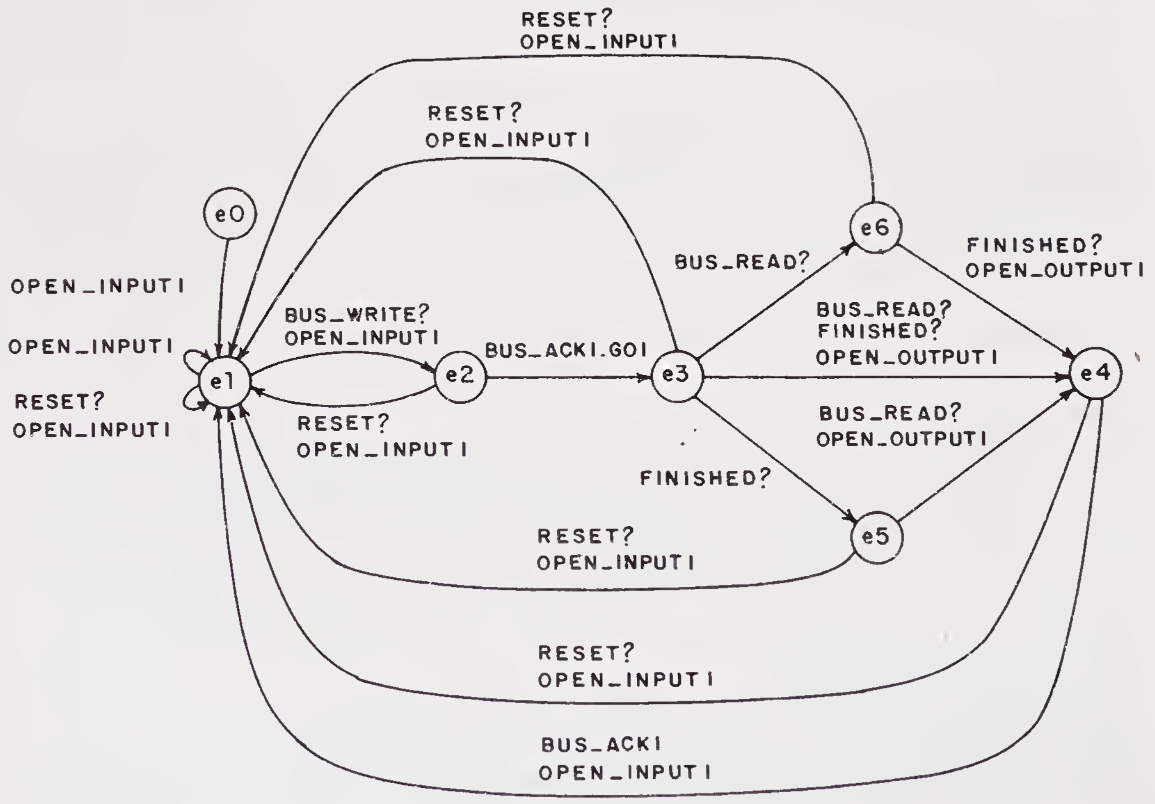


Figure A1. The interface automaton.

Notice also the reset arrows that go from any state into state 1: they are all generated by the single "loop ··· each RESET" statement. Of course, such a small automaton can be easily designed by hand. The advantage of ESTEREL programming really appears for more complex controllers. The modularity of the language, its built-in concurrency, and the power of its control structures allow the user to build controllers by assembling individually simple modules into bigger ones. For example, to perform speed benchmarks on PERLE0, we use a variant of the bus interface that inputs two data words and performs computation and output twice in a row. To obtain this interface, one just changes the Interface module body into (roughly).

```
run Input [signal OPEN_INPUT_1/OPEN_INPUT];
run Input [signal OPEN_INPUT_2/OPEN_INPUT];
run ComputeAndOutput [signal OPEN_OUTPUT_1/OPEN_OUTPUT,
  GO_1/GO];
run ComputeAndOutput [signal OPEN_OUTPUT_2/OPEN_OUTPUT,
  GO_2/GO]
```

Usually, a relatively simple change to a specification involves a simple and local change to an ESTEREL program. This is definitely not true of finite automata, which are highly unstable w.r.t. specification changes. We strongly believe that programming controllers in ESTEREL is one order of magnitude simpler that designing finite state machines by hand.

References

Berry G 1989 *Real-time programming: general purpose or special-purpose languages, Information Processing 1989* (ed.) G X Ritter (Amsterdam: Elsevier Science/North Holland) pp. 11–17

Berry G, Benveniste A 1991 The synchronous approach to reactive and real-time systems. Another look at real time programming. *Proc. IEEE* 79: 1270–1282

Berry G, Cosserat L 1984 The synchronous programming languages Esterel and its mathematical semantics. In *Seminar on concurrency. Lecture Notes in Computer Science Vol. 197* (eds) S Brookes, G Winskel (Berlin: Springer Verlag) pp. 389–448

Berry G, Couronne P, Gonthier G 1988 Synchronous programming of reactive systems: an introduction to Esterel. In *Programming of future generation computers* (eds) K Fuchi, M Nivat (Amsterdam: Elsevier Science/North Holland) pp. 35–55

Berry G, Gonthier G 1989 The Esterel synchronous programming language: Design, semantics, implementation, Res. Rep. 842, to appear in *Science of computer programming*

Berry G, Gonthier G 1991 Incremental development of an HDLC entity in Esterel. *Comput. Networks* 22: 35–49

Berthet C, Coudert O, Madre J -C 1990 New ideas on symbolic manipulations of finite state machines. In *Proc. International Conference on Computer Design* (*ICCD*)

Bertin P, Roncin D, Vuillemin J 1989 Introduction to programmable active memories. In *Systolic array processors* (eds) J McCanny, J McWhirter, E Swartzlander (Englewood Cliffs, NJ: Prentice-Hall) pp. 301–309

Boudol G, Roy V, de Simone R, Vergamini D 1990 Process calculi. from theory to practice: verification tools. In *Automatic verification methods for finite state systems. Lecturer Notes in Computer Science. Vol. 407* (Berlin: Springer Verlag) pp. 1–10

Boussinot F, de Simone R 1991 The Esterel language. Another look at Real Time Programming. *Proc. IEEE* 79: 1293–1304

Brayton R K, Hachtel G D, Sangiovanni-Vincentelli A L 1990 Multilevel logic synthesis. *Proc. IEEE* 78: 264–300

Caspi P, Halbwachs N, Pilaud D, Plaice J 1987 Lustre: a declarative language for programming synchronous systems. In *Proc. 14th Annual ACM Symposium on Principles of Programming Languages, Munich*, pp. 177–188

Clarke E M, Long D E, McMillan K L 1991 A language for compositional specification and verification of finite state hardware controllers. Another look at real time programming. *Proc. IEEE* 79: 1283–1292

Clement D, Incerpi J 1989 Programming the behaviour of graphical objects using Esterel. In *TAPSOFT'89. Lecture Notes in Computer Science. Vol. 352* (Berlin: Springer Verlag)

Coudert O, Madre J -C 1990 A unified framework for the formal verification of sequential circuits. In *Proc. of International Conference on Computer Aided Design* (*ICCAD*) Santa Clara, USA

Cousineau G 1980 An algebraic definition for control structure. *Theor. Comput. Sci.* 12: 175–192

Gauthier T, Le Guernic P, Besnard L 1987 Signal, a declarative language for synchronous programming of real-time systems. In *Proc. 3rd Conf. on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science. Vol. 274* (Berlin: Springer Verlag)

Gonthier G 1988 *Sémantique et Modeles d'Execution des languages Réactifs Synchrones; Application ă Esterel*, There d'Informatique, Universite d'Orsay

Halbwachs N, Caspi P, Pilaud D 1991 The synchronous dataflow programming language Lustre. Another look at real time programming. *Proc. IEEE* 79:

Kahn G 1988 Natural semantics. In *Programming of future generation computers* (eds) K Fuchi, M Nivat (Amsterdam: North Holland) pp. 237–258

Le Guernic P, Le Borgne M, Gauthier T, Le Maire C 1991 Programming real time applications with signal. Another look at real time programming. *Proc. IEEE* 79:

Mejia Olvera M C 1989 *Contribution a la Conception d'un Réseau Local Temps Récl pour la Robotique*, These de Docteur-Ingenieur, Universite de Rennes

Murakami G, Sethi R 1990 Terminal Call Processing in Esterel, Research Report 150, AT & T Bell Laboratories

Savoj H, Touati H, Brayton R K 1991 The use of image computation techniques in extracting local don't cares and network optimization. In *Proceedings IEEE International Conference on Computer-Aided Design* (*ICCAD*), pp. 514–517

Shand M, Bertin P, Vuillemin J 1990 Hardware speedups in long integer multiplication. In *Proc. 2nd Annual ACM Symposium on Parallel, Algorithms and Architectures*, pp. 138–145

Sādhanā, Vol. 17, Part 1, March 1992, pp. 131–165. © Printed in India.

Models and logics for true concurrency

KAMAL LODAYA[1], MADHAVAN MUKUND[2], R RAMANUJAM[1] and P S THIAGARAJAN[2]
[1]The Institute of Mathematical Sciences, Taramani, CIT Campus, Madras 600 113, India
[2]School of Mathematics, SPIC Science Foundation, 92 G.N. Chetty Road, T. Nagar, Madras 600 017, India

Abstract. A distributed computer system consists of different processes or *agents* that function largely autonomously and coordinate their actions by communicating with each other. In such a situation, actions may be performed by different agents of the system locally, in a concurrent manner.

In this paper, we first discuss formal models of distributed systems in which concurrency is specified *explicitly*, in contrast to more traditional approaches where concurrency is represented *implicitly* as a nondeterministic choice between all possible sequentializations of concurrent actions. This naturally leads to models based on partially-ordered sets of actions rather than sequences of actions and is often called the *true concurrency* approach. The models we focus on are distributed transition systems, elementary net systems and event structures.

In the second half of the paper, we develop a family of logics to specify and reason about the behavioural properties of the models we have described. The logics we define are extensions of temporal logic with new modalities to directly describe concurrency.

This paper is essentially a survey of work done by the authors during the last few years on modelling distributed systems with true concurrency and using logic to reason about these models. The emphasis is on motivating definitions through examples and on presenting major results, without going into too many formal details. We provide pointers to the literature where these details can be found.

Keywords. Concurrency; temporal logic; distributed systems; logics of programs.

1. Introduction

The study of distributed systems and computations is an important topic of research in computer science. A distributed system consists of a number of essentially autonomous components that work together to perform a complex task.

A computer network which brings together a heterogeneous collection of computing resources and users dispersed over a wide geographic area is a classic example of a distributed system. Distributed databases constitute yet another class of examples. At a lower level, computer protocols which facilitate efficient and reliable transmission of electronic data and operating systems which coordinate the activities of multiple processes (programs) in the presence of multiple processors can also be viewed as distributed systems. With the advent of VLSI systems, the notion of a distributed system is also becoming relevant at the circuit level.

The *theory* of distributed systems consists of formulating abstract mathematical models of distributed systems and studying the properties of these models. A basic motivation in the study of formal models is to develop tools and techniques using which one can specify, analyse and implement distributed systems. Another goal is to develop formal means for reasoning about the behaviour of distributed systems. This is important because one would like to ensure that a specification is in some sense consistent before one attempts an analysis or an implementation. Even more importantly, one would like to guarantee that a proposed implementation indeed meets the requirements of a specification.

In this paper, we present some of our work in the last few years on modelling distributed systems with true concurrency, using logic to reason about these models. The emphasis is on motivating definitions through examples and on presenting major results. No attempt will be made to go into formal details; we shall provide pointers to the literature where these details can be found.

In the first part of the paper, we introduce three models called distributed transition systems, elementary net systems and event structures. Using these models, we illustrate some of the fundamental features of distributed systems, such as causality, choice and concurrency.

In the second half of the paper, we develop a family of logics to specify and reason about the behavioural properties of the models considered in the first half of the paper.

2. Models for true concurrency

Typically, a distributed system consists of spatially separated processes or agents performing a joint task. The agents function largely autonomously and coordinate their actions by communicating with each other. In such a situation, actions may be performed by different agents of the system locally, in a concurrent manner.

Informally, we say that two events are concurrent if they occur with no *a priori* ordering over their occurrences. This is in contrast to a sequential system in which any two events that occur in a computation must be ordered.

In addition to concurrency, two other aspects are of interest in the theory of distributed systems – causality and choice. Causality refers to the fact that certain events in a distributed system can only occur in a fixed order; for example, a message can be received only after it has been sent. The receipt of a message is said to be causally dependent on the sending of the message.

Choice captures the fact that systems can behave in an indeterminate fashion. In other words, at certain points of the computations, the system may choose between alternative events, leading to different behaviours.

As we shall see, labelled transition systems are simple and convenient models of sequential systems which can explicitly describe causality and choice, but which do not have a natural way of representing concurrency. One way of describing concurrency in the framework of transition systems is in terms of indeterminacy. In this approach, the fact that a set of actions may be performed concurrently is represented by permitting the system to choose between all possible sequentializations of the actions. This approximation of concurrency by *interleaving* is used in various algebraic approaches to the theory of distributed systems such as a calculus for communicating systems (CCS) (Milner 1989), communicating sequential processes (CSP) (Hoare 1984) and algebra of communicating processes (ACP) (Bergstra & Klop 1984).

Such an implicit representation of concurrency leads to problems in analysing system behaviour, due to the combinatorial explosion in the number of possible interleavings. We follow an alternative approach, called "true concurrency", where concurrency is represented explicitly in the models.

Many abstract models of distributed systems have been suggested which explicitly deal with the phenomena of causality, choice and concurrency. Here, we shall consider three of these models – distributed transition systems, elementary net systems and event structures. We shall also discuss a model called communicating sequential agents. This model, based on a restricted class of event structures, captures in a natural way the intuitive picture of a distributed system as a collection of sequential agents coordinating their actions through communication.

2.1 *Distributed transition systems*

Before discussing models of concurrent systems, let us briefly look at sequential systems. Transition systems are a basic model of sequential systems.

DEFINITION 1.1

A (Σ-*labelled*) *transition system* is a triple $TS = (S, \Sigma, \rightarrow)$ where

(1) S is a set of states.
(2) Σ is a set of actions.
(3) $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.

If $(s, a, s') \in \rightarrow$, then the idea is that the action a can occur at state s and after the execution of a the system assumes the state s'. We shall often write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \rightarrow$.

Figure 1 is a graphical representation of a transition system. The nodes of the graph represent the states of the system. The edges, labelled by actions from Σ, reflect the transition relation $\rightarrow$.

Clearly the structure of a transition system captures both the basic phenomena present in sequential systems – causality and choice. The transition relation can be used to determine the causal dependencies between system states. Choice is specified

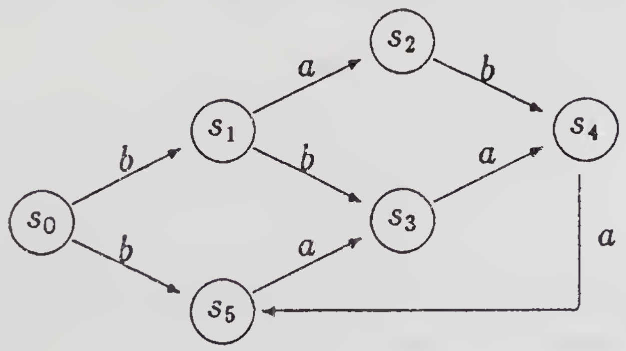


Figure 1. A transition system.

by branching in the transition system. In other words, if $s \xrightarrow{a} s'$ and $s \xrightarrow{b} s''$ both belong to the transition relation, then the system at state s can choose between the actions a and b. For example, at s_1 the system shown in figure 1 can either move by an a to s_2 or move by a b to s_3. In general, different choices available to the system at a state may be labelled by the *same* action. In other words, the behaviour could be nondeterministic. For instance, at s_0 this system can move on b either to s_5 or to s_1.

In this example, starting at s_1, either the action a can occur followed by the action b or the action b can occur followed by the action a. In the interleaving approach to concurrency, this situation often amounts to saying that a and b can occur concurrently at s_1.

However, we would like to maintain a clear distinction between nondeterminism and concurrency. Hence, to describe concurrency in a transition system, we enrich the relation $\rightarrow$ by permitting a transition to be labelled by a finite set of actions from Σ, rather than just by a single action. Thus, we will now have elements in $\rightarrow$ of the form $s \xrightarrow{u} s'$, where u is a finite subset of Σ. The idea is that the actions in u can occur at s with no order over their occurrences. When they have all occurred, the resulting state is s'. The set of actions u is termed a *concurrent step*.

Henceforth, given a set X, $\wp(X)$ denotes the set of subsets of X and $\wp_{fin}(X)$ denotes the set of *finite* subsets of X. We can now formally define distributed transition systems as follows.

DEFINITION 1.2

A distributed transition system (DTS) is a triple $DTS = (S, \Sigma, \rightarrow)$ where

(1) S is a set of states;
(2) Σ is a set of actions;
(3) $\rightarrow \subseteq S \times \wp_{fin}(\Sigma) \times S$ is the step transition relation satisfying for all s, s' in S:
  (a) $s \xrightarrow{\emptyset} s'$ iff $s = s'$.
  (b) for all $u \in \wp_{fin}(\Sigma)$, if $s \xrightarrow{u} s'$ then there exists a function $f: \wp(u) \rightarrow S$ such that $f(\emptyset) = s$, $f(u) = s'$ and for every v_1, $v_2 \in \wp(u)$ with $v_1 \subseteq v_2$, it is the case that $f(v_1) \xrightarrow{v_2 - v_1} f(v_2)$.

We often say that $DTS = (S, \Sigma, \rightarrow)$ is a DTS over Σ. For convenience, we write $s \xrightarrow{a} s'$ instead of $s \xrightarrow{\{a\}} s'$.

The new definition of $\rightarrow$ is a bit involved because we have to ensure that any nontrivial "substep" of a concurrent step is also performed as a concurrent step. The function f in clause (3.b) is said to define a *u-cube* (from s to s'). The existence of the u-cube guarantees that the mutual independence of the actions in u holds for all the substeps as well. For example, figure 2 shows the cube generated by a concurrent step consisting of three events. To avoid cluttering up the figure, "interior" arrows such as $f_\emptyset \xrightarrow{\{a,b\}} f_{ab}$ and $f_b \xrightarrow{\{a,c\}} f_{abc}$ have not been drawn.

Figure 3 is an example of a distributed transition system modelling the allocation of a shared resource to different processes within a system. In the example, we have 3 processes P_1, P_2 and P_3 functioning in an operating environment that supports multiprocessing. The resource – say, for example, blocks of memory – is available in "units". There are totally 5 units available. The 3 processes require 2, 3 and 5 units, respectively, of the resource at a time. In this DTS, $\Sigma = \{a_1, a_2, a_3, r_1, r_2, r_3\}$, where a_i denotes that process P_i has been allocated the entire amount of the resource that it needs and r_i denotes that P_i has released the resource it has been allocated. The states

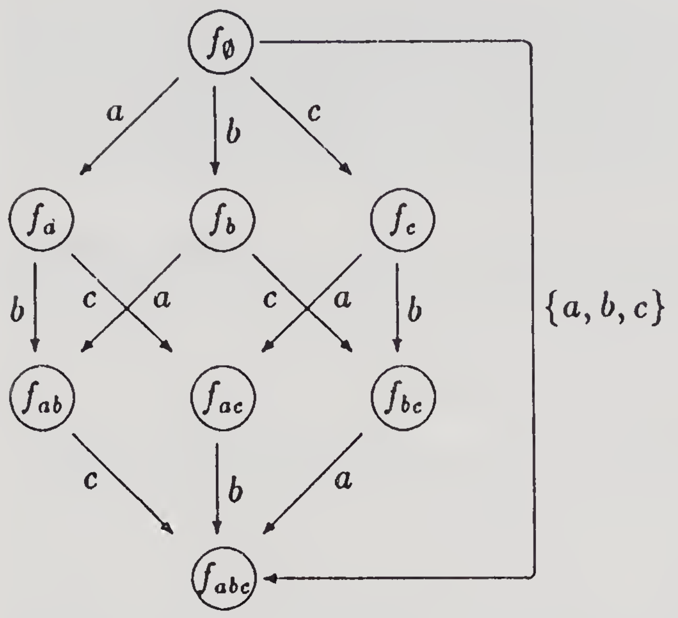


Figure 2. A "cube" generated by a concurrent step.

of the DTS are ordered pairs consisting of the number of unallocated units of the resource available in the system along with the set of processes currently in possession of their required quota of the shared resource.

Thus, at the state $(5, \emptyset)$, no processes are active and all 5 units of the resource are available. At this state, the system can either allocate units of the resource to one of the three processes, or perform a concurrent step allocating resources to both P_1 and P_2. Notice that the transition from $(3, \{P_1\})$ to $(2, \{P_2\})$ can either be performed as a concurrent step $\{a_2, r_1\}$ or by interleaving the two actions. In one interleaving, however, $(5, \emptyset)$ is reached as an intermediate state, at which point the resource can be allocated to P_3 instead of P_2. Thus, in this case, the effect of the interleavings is not quite the same as that of the concurrent step.

In general, it is important to note that clause (3.b) in definition 1.2 is merely an implication. The existence of a function from $\wp(u)$ into S which fulfills the stated

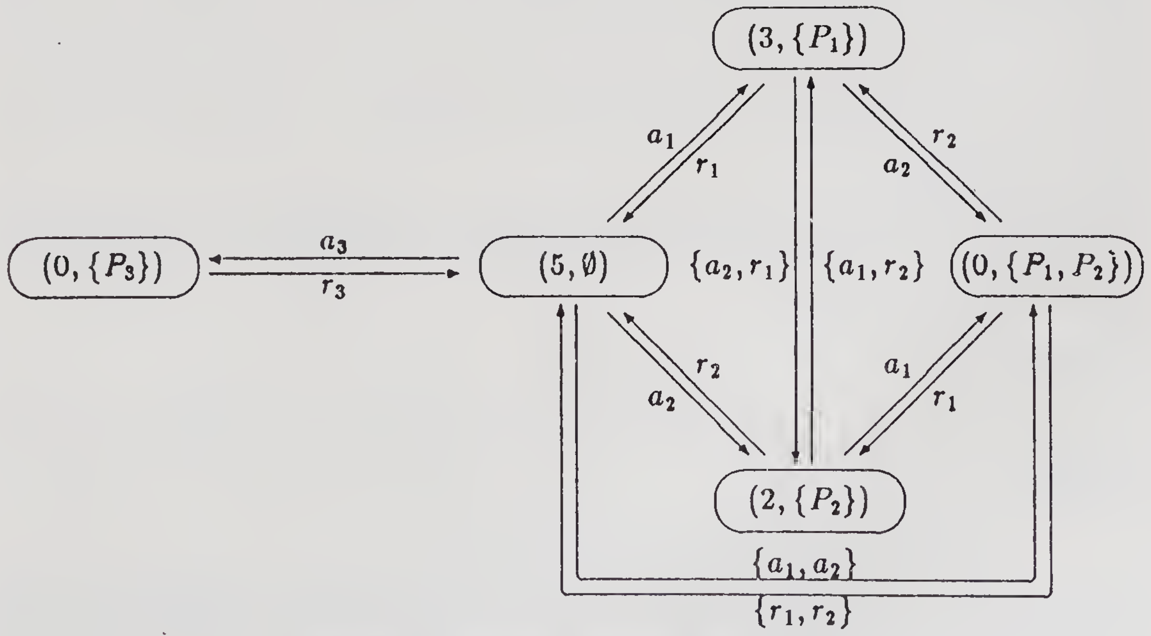


Figure 3. A distributed transition system.

requirements does not guarantee the existence of a concurrent step. This is in line with our philosophy that concurrency should be clearly differentiated from interleaving. As we have seen above, interleavings may permit unintended deviations from the behaviour expected of a concurrent step. In fact, it is possible to have a concurrent step as well as an interleaving of the step performed at a state but leading to two different states.

Finally, we introduce the important notion of reachability in a transition system. Given $TS = (S, \Sigma, \rightarrow)$ we define $\mathscr{R}(s_0)$, the *reachability set* of $s_0 \in S$, as the least subset of S containing s_0 satisfying:

$$\text{If } s \in \mathscr{R}(s_0),\ a \in \Sigma \text{ and } s \xrightarrow{a} s', \text{ then } s' \in \mathscr{R}(s_0).$$

Thus, $\mathscr{R}(s_0)$ is the set of states reachable from s_0 in a finite number of steps using $\rightarrow$.

2.2 *Elementary net systems*

In a distributed transition system, concurrency is explicitly introduced into a transition system by permitting transitions between states via finite sets of actions called concurrent steps. In effect, the notion of a state is left unchanged and the transition relation is enriched to model concurrency.

An alternative way of introducing concurrency into a transition system is to "distribute" the states of the system. The states of a DTS correspond to the global states of the concurrent system being modelled by the DTS. Rather than regard these global states as indivisible entities, we can break them up into atomic components which can be regarded as the local states of the different processes within the system. The global states of the system can then be characterized in terms of the local states.

By distributing the states of the model in this manner, we can clearly distinguish concurrency from choice without having to define a transition relation involving sets of actions as in a DTS. Instead, the transition relation is designed to capture the fact that the change of state accompanying each event occurrence in the system is "localized" to those processes that actually participate in the event. As a result, when an event occurs, only specific local components of the global state are affected, leaving the rest of the components untouched. Thus, two events that are enabled at a global state of the system can occur concurrently if the local states that they affect are disjoint. On the other hand, if the local states affected by the two events overlap, they cannot both occur in the same computation at that state and so a choice must be made between them.

Net theory deals with models of concurrent systems based on this approach. Here we describe elementary net systems, which are a basic model in this theory. We begin with the definition of a net.

DEFINITION 2.1

A *net* is a triple $N = (B, E, F)$ where B and E are disjoint sets and $F \subseteq (B \times E) \cup (E \times B)$ satisfies:

$$\forall x \in B \cup E : \exists y \in B \cup E : (x, y) \in F \text{ or } (y, x) \in F.$$

The elements or B are called *conditions* and are used to denote atomic local states. The elements of E are called *events* and are used to represent atomic actions. The *flow relation F* models a *fixed* neighbourhood relation between the conditions and events of a system. This flow relation determines the way in which the atomic actions affect the atomic local states. The restriction on F in the definition of a net ensures that there are no isolated conditions or events.

We can now define an elementary net system as follows.

DEFINITION 2.2

An *elementary net system* is a quadruple $\mathcal{N} = (B, E, F, c_{in})$ where

(1) $N_{\mathcal{N}} = (B, E, F)$ is a net called the underlying net of $\mathcal{N}$.
(2) $c_{in} \subseteq B$ is the *initial case.*

Figure 4 is an example of an elementary net system. We have used the conventional graphical notation for nets – conditions are represented by circles, events by boxes and the flow relation by directed arcs. The "marked" conditions denote the initial case c_{in}.

For e in E the conditions "pointing into" e via F are called the *pre-conditions of* e and are denoted by ${}^{\bullet}e$. Similarly, the conditions "pointing away" from e via F are called the *post-conditions of* e and are denoted by $e^{\bullet}$. More formally we have

$$
\begin{aligned}
{}^{\bullet}e &\overset{\text{def}}{=} \{b \mid (b, e) \in F\},\\
e^{\bullet} &\overset{\text{def}}{=} \{b \mid (e, b) \in F\}.
\end{aligned}
$$

A state of a net system, called a *case*, consists of a set of conditions $c \subseteq B$. The conditions in c are said to *hold* when the system is at the case c. Thus, c_{in} is the set of conditions that hold when the system starts up.

The system moves from one case to another through the occurrence of events from E. An event can occur at a case iff all its pre-conditions hold and none of its post-conditions do at the case. When an event occurs all its pre-conditions cease to hold and all its post-conditions begin to hold.

In graphical terms, an event e can occur at a case c iff all the conditions pointing into e are "marked" at c and none of the conditions pointing away from e are. For

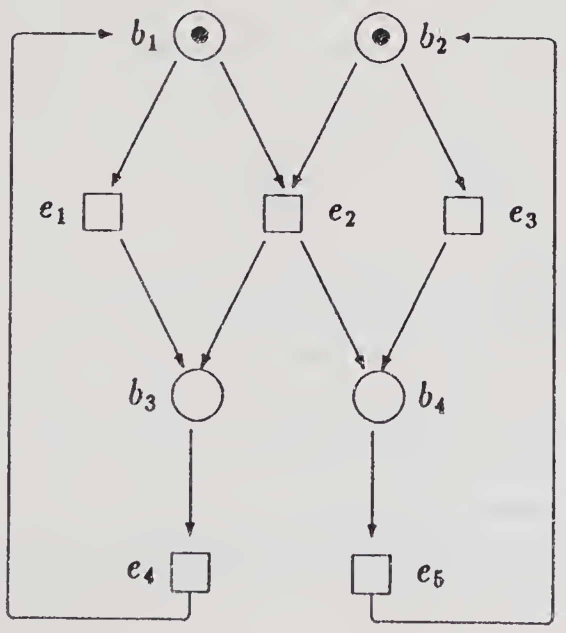


Figure 4. An elementary net system.

example, in figure 4, the event e_1 can occur at the initial case $c_{in} = \{b_1, b_2\}$. When e_1 occurs, we "unmark" all the pre-conditions of e_1 and "mark" all its post-conditions, leaving the other conditions in c_{in} untouched. Thus, after the occurrence of e_1, the system is at the case $\{b_2, b_3\}$.

We can formalise this by defining $\rightarrow_N \subseteq \wp(B) \times E \times \wp(B)$, the elementary transition relation generated by the net $N = (B, E, F)$ as follows.

$$\rightarrow_N = \{(x, e, x') | x - x' = {}^\bullet e, \quad x' - x = e^\bullet\}$$

Using this transition relation, we can associate a transition system with an elementary net system as follows.

DEFINITION 2.3

Let $\mathcal{N} = (B, E, F, c_{in})$ be a net system.

(1) $C_{\mathcal{N}}$, the *state space* of $\mathcal{N}$, is the least subset of $\wp(B)$ containing c_{in} such that if $c \in C_{\mathcal{N}}$ and $(c, e, c') \in \rightarrow N_{\mathcal{N}}$ then $c' \in C_{\mathcal{N}}$.
(2) $TS_{\mathcal{N}} = (C_{\mathcal{N}}, E, \rightarrow_{\mathcal{N}})$ is the *transition system associated with* $\mathcal{N}$, where $\rightarrow_{\mathcal{N}}$, is $\rightarrow_{N_{\mathcal{N}}}$ restricted to $C_{\mathcal{N}} \times E \times C_{\mathcal{N}}$.

For the net system shown in figure 4, $\{\{b_1, b_2\}, \{b_1, b_4\}, \{b_2, b_3\}, \{b_3, b_4\}\}$ is its state space.

Let $\mathcal{N} = (B, E, F, c_{in})$ be a net system with $c \in C_{\mathcal{N}}$ and $e \in E$. Then e is said to be enabled at c – denoted $c[e\rangle$ – iff there exists $c' \in C_{\mathcal{N}}$ such that $c \xrightarrow{e} c'$, where as usual $c \xrightarrow{e} c'$ abbreviates $(c, e, c') \in \rightarrow_{\mathcal{N}}$.

As we had mentioned at the beginning of this section, we can clearly separate concurrency from choice once we have distributed the global states of a transition system into local components.

Let $\mathcal{N} = (B, E, F, c_{in})$ be a net system and $e, e' \in E$. We say that e and e' can occur concurrently at a case c – denoted $c[\{e, e'\}\rangle$ – iff $c[e\rangle$ and $c[e'\rangle$ and $({}^\bullet e \cup e^\bullet) \cap ({}^\bullet e' \cup e'^\bullet) = \emptyset$. Thus, e and e' can occur concurrently at a case if they can occur individually and their "neighbourhoods" are disjoint.

Similarly we can define the notion of conflict. Let $\mathcal{N}$ be a net system as above with $e, e' \in E$. e and e' are said to be in conflict at a case c iff $c[e\rangle$ and $c[e'\rangle$ but not $c[\{e, e'\}\rangle$. Thus, if e and e' are in conflict at c, it means that they are both individually enabled at c, but they cannot occur together at c. For the computation to proceed, the conflict must be resolved by making a (nondeterministic) choice between the two events.

The definition of $\rightarrow_{\mathcal{N}}$ is designed to ensure that the notion of change of state in an elementary net system is fairly restricted.

First, notice that an event must cause the same change in the system state whenever it occurs; its pre-conditions cease to hold and its post-conditions begin to hold. Thus, if $c_1 \xrightarrow{e} c_2$ and $c_3 \xrightarrow{e} c_4$ are both possible in a net system, then it must be the case that $c_1 - c_2 = c_3 - c_4 = {}^\bullet e$ and $c_2 - c_1 = c_4 - c_3 = e^\bullet$.

Further, to determine whether an event e is enabled at a case c, it is sufficient to look at the conditions contained in ${}^\bullet e$ and $e^\bullet$. e is enabled at c iff ${}^\bullet e \subseteq c$ and $e^\bullet \cap c = \emptyset$ – no "side-conditions" are involved in the enabling of an event.

Finally, it turns out that the transition system $TS_{\mathcal{N}}$ associated with a net system $\mathcal{N}$ is deterministic; that is, $c \xrightarrow{e} c'$ and $c \xrightarrow{e} c''$ implies that $c' = c''$. To connect up

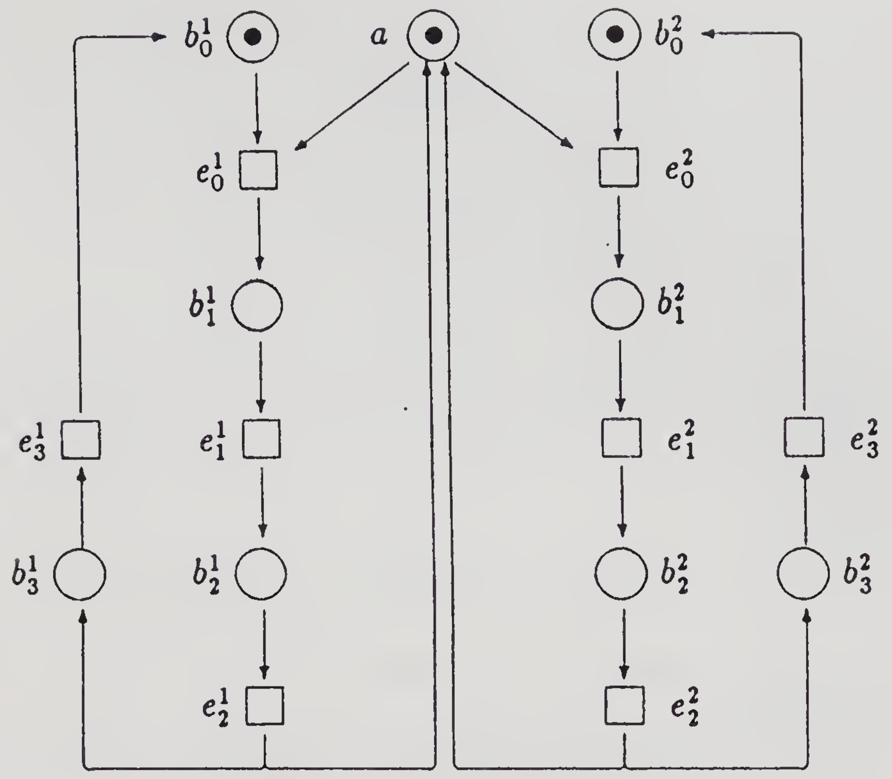


Figure 5. Mutual exclusion.

with other approaches to the theory of distributed systems, nondeterminism can be introduced into $TS_{\mathcal{N}}$ by labelling the events in E. We shall come back to this point later in this section.

Let us consider an example of modelling a distributed system using an unlabelled elementary net system. Consider the problem of sharing resources in a distributed system. Suppose that there are two processes P_1 and P_2 in the system which require access to a common resource r. Suppose that r can be used by only one process at a time – r could, for instance, be a printer. Then, when one of the processes is granted access to r, the other process should be prevented from accessing r till the first process releases it. This will ensure that at any state during a computation of the system, at most one process can actually be using that resource.

Figure 5 models a solution to this problem of mutual exclusion. In this net system the process P_i, $i = 1, 2$, is represented by the conditions $\{b_0^i, b_1^i, b_2^i, b_3^i\}$ and the events $\{e_0^i, e_1^i, e_2^i, e_3^i\}$. Each process is modelled as a simple loop consisting of four events – getting access to r (e_0^i), utilizing r (e_1^i), releasing r (e_2^i) and performing some internal computations not involving r (e_3^i). At the initial case, both processes are waiting for access to r. The additional condition a functions as an arbitrator which enforces mutual exclusion of access to r. For example, suppose that e_0^2 occurs initially, giving P_2 access to r. Since a ceases to hold e_0^1 is no longer enabled. Thus, P_1 can gain access to r only after P_2 releases r by the occurrence of e_2^2. It is easy to check that b_1^1 and b_1^2 can never hold together in this net system. On the other hand, the conditions b_3^1 and b_3^2 can hold at the same case – that is, the events e_3^1 and e_3^2 which do not involve the use of r can occur concurrently in this system.

Finally, we show that we can describe the behaviour of elementary net systems in terms of distributed transition systems. Consider an elementary net system $\mathcal{N} = (B, E, F, c_{in})$. The transition system $TS_{\mathcal{N}}$ contains information about the causality and conflict present in $\mathcal{N}$. To describe the concurrency present in $\mathcal{N}$, it is sufficient to augment $TS_{\mathcal{N}}$ with additional transitions labelled by concurrent steps, as follows.

We first extend the notion of a pair of events being concurrently enabled at a case to a set of events. Let $u=\{e_1,e_2,\ldots,e_n\}$ be a finite subset of E. We say that u is concurrently enabled at a case $c\in C_{\mathcal{N}}$ – denoted $c[u\rangle$ – iff $c[e_i\rangle$ for each $e_i\in u$ and, further, $c[\{e_1,e_2\}\rangle$ for every pair of distinct events $e_1,e_2\in u$.

We can then define the step transition relation $\Rightarrow_{\mathcal{N}}$ as follows,

$$\Rightarrow_{\mathcal{N}}=\{(c,u,c')\mid c,c'\in C_{\mathcal{N}},\ c[u\rangle \text{ and } c-c'={}^{\bullet}u,\ c'-c=u^{\bullet}\}.$$

Here ${}^{\bullet}u$ and $u^{\bullet}$ denote the unions of the pre-conditions and post-conditions of the events contained in u. Note that $\rightarrow_{\mathcal{N}}$ is "included" in $\Rightarrow_{\mathcal{N}}$ in the sense that if $(c,e,c')\in\rightarrow_{\mathcal{N}}$ then $(c,\{e\},c')\in\Rightarrow_{\mathcal{N}}$. We can then immediately establish the following.

PROPOSITION 2.4.

$DTS_{\mathcal{N}}=(C_{\mathcal{N}},\ E,\ \Rightarrow_{\mathcal{N}})$ is a distributed transition system over E.

It is easy to verify that the concurrency and choice present in $\mathcal{N}$ is precisely captured by the DTS ($DTS_{\mathcal{N}}$).

However, notice that this DTS is deterministic, for the same reason that the transition system $TS_{\mathcal{N}}$ is. As we had mentioned earlier, we can introduce nondeterminism by labelling the events.

DEFINITION 2.5.

A *Σ-labelled elementary net system* is a pair $\mathcal{N}_{\Sigma}=(\mathcal{N},\phi)$, where $\mathcal{N}=(B,E,F,c_{in})$ is an elementary net system, called the underlying net system of $\mathcal{N}_{\Sigma}$. Σ is a set of labels and $\phi\colon E\rightarrow\Sigma$ is the labelling function.

The notions we have developed for net systems can be transported to labelled net systems in the obvious way. To represent the behaviour of a labelled net system $\mathcal{N}_{\Sigma}$ as a DTS, we can define $DTS_{\mathcal{N}_{\Sigma}}$ to be the DTS over Σ obtained by using the labelling function ϕ to rename the actions in $DTS_{\mathcal{N}}$, the DTS over E generated by the underlying net system $\mathcal{N}$.

However, in general we need to place a restriction on the labelling function in order to get a neat translation from labelled net systems to distributed transition systems. In a DTS, we have restricted concurrent steps to be *sets* of actions. On the other hand, a labelled net system $\mathcal{N}_{\Sigma}$ may generate a concurrent step in $DTS_{\mathcal{N}_{\Sigma}}$ where two distinct events in the step have the *same* label. To avoid dealing with multisets in concurrent steps that arise in this fashion, we require that events which can occur concurrently in the underlying net system $\mathcal{N}$ have distinct labels.

Let $\mathcal{N}_{\Sigma}=(B,E,F,c_{in},\phi)$ be a Σ-labelled net system. The labelling function ϕ is said to be *co-injective* if it satisfies the following condition.

$$\forall e_1,e_2\in E{:}(\exists c\in C_{\mathcal{N}}{:}c[\{e_1,e_2\}\rangle) \text{ implies } \phi(e_1)\neq\phi(e_2).$$

PROPOSITION 2.6.

Let $\mathcal{N}_{\Sigma}=(\mathcal{N},\phi)$ be a Σ-labelled elementary net system, where $\mathcal{N}=(B,E,F,c_{in})$, such that ϕ is co-injective. Then $DTS_{\mathcal{N}_{\Sigma}}=(C_{\mathcal{N}},\Sigma,\Rightarrow_{\mathcal{N}_{\Sigma}})$ is a DTS over Σ, where

$$\Rightarrow_{\mathcal{N}_{\Sigma}}=\{(c,\phi(u),c')\mid(c,u,c')\in\Rightarrow_{\mathcal{N}}\}.$$

2.3 *Event structures*

To reason about the behaviour of a distributed transition system or an elementary net system, we have to examine all the computations of the underlying "machines" defined by the model. For this, it is convenient to work with an abstract representation of the *entire* behaviour of the system. This behavioural description should include information about all the computations of the system, explicitly identifying the causal dependencies and concurrency present within each computation. In addition, it should also have a way of describing the branching points in the system behaviour.

Before discussing behavioural representations of concurrent systems, let us first go back to sequential transition systems. A computation of a sequential transition system $TS=(S,\Sigma,\rightarrow)$ starting at some state $s_0 \in S$ is an alternating sequence of actions and states which obeys the transition relation $\rightarrow$. We shall restrict our attention to the *maximal* computations of the system – those that cannot be extended by performing any more actions. Thus, a maximal computation is a finite sequence just in case a state is reached at the end of the sequence from which no transition is possible; otherwise, it is an infinite sequence.

A natural way to group together the sequences which correspond to computations of $TS=(S,\Sigma,\rightarrow)$ starting from s_0 is in the form of a tree. The nodes of the tree are labelled by states from S and the edges are labelled by actions from Σ. The root node is labelled by the initial state s_0. Each maximal path in the tree now corresponds to a computation of the system. The branching points in the tree are the states where the system makes choices between different possible actions.

In the case of models exhibiting concurrency, the situation is more complicated. A computation of such a system is a partially ordered set of actions, not a simple sequence, so we need a more sophisticated method of collecting all the computations together in a single structure. An elegant way of achieving this is to use event structures. Event structures are behavioural models of distributed systems in which causality, concurrency and choice (conflict) are represented explicitly.

Prime event structures, introduced in Nielsen *et al* (1980), are the simplest type of event structures. They have a rich theory and are closely related to both net systems and domains. Since we deal only with prime event structures in this paper, henceforth we shall simply call them event structures.

DEFINITION 3.1

An *event structure* is a triple $ES=(E,\leqslant,\#)$ where

(1) E is a set of *event occurrences.*
(2) $\leqslant \subseteq E \times E$ is a partial order called the *causality relation.*
(3) $\# \subseteq E \times E$ is an irreflexive and symmetric *conflict relation.*
(4) $\#$ is inherited via $\leqslant$ in the sense that $e_1 \# e_2 \leqslant e_3$ implies that $e_1 \# e_3$ for every e_1, e_2, e_3 in E.

An element of E represents the occurrence of an event within a specific context. Thus, if the same event can occur in different contexts, "copies" of it will be present in the event structure. This is why we have called the elements of E event occurrences rather than events.

If $e_1 \leqslant e_2$, then e_2 is causally dependent on e_1. Thus, in any computation of the system, e_2 can occur only if e_1 has already occurred. As usual we let $\geqslant$ stand for $\leqslant^{-1}$.

The # relation identifies pairs of events which are inconsistent with each other and hence cannot both occur during the same computation. The last clause of definition 3.1 ensures that if $e_1 \# e_2$ then events that are causally dependent on e_1 are in conflict with events that are causally dependent on e_2 – in other words, the inconsistency of e_1 and e_2 is inherited by events that follow these two events.

Two events that are neither causally related nor in conflict with each other can both occur within a computation with no order over their occurrence. We can thus define the concurrency relation *co* in an event structure $ES = (E, \leqslant, \#)$ in terms of $\leqslant$ and # as follows.

$$co \overset{\text{def}}{=} E \times E - (\leqslant \cup \geqslant \cup \#).$$

Notice that *co*, like #, is irreflexive and symmetric. Clearly, every pair of *distinct* events in an event structure belongs to exactly one of the four relations $\{\leqslant, \geqslant, \#, co\}$.

It is useful to define one more auxiliary relation. Let $ES = (E, \leqslant, \#)$ be an event structure and $e, e' \in E$. Then

$$e \#_\mu e' \overset{\text{def}}{=} e \# e' \text{ and } \forall e_1, e'_1 \in E: [e_1 \leqslant e \text{ and } e'_1 \leqslant e' \text{ and } e_1 \# e'_1 \text{ implies } e_1 = e \text{ and } e'_1 = e'].$$

$\#_\mu$ identifies the minimal elements (under $\leqslant$) of the # relation and is hence called the minimal conflict relation. $\#_\mu$ identifies the actual branching points in the behaviour where choices are made between conflicting events. This "basic" conflict then propagates to causally related events and "generates" other conflicts.

Figure 6 is an example of an event structure. The squiggly lines represent the $\#_\mu$ relation. The causality relation is shown in the form of the associated Hasse diagram. The # relation is then uniquely determined by the last part of definition 3.1. In this event structure, $e_1 \# e_6$ because $e_1 \#_\mu e_2 \leqslant e_6$. It is also easy to see that $e_6 co e_7$.

The states of an event structure are called configurations. A configuration identifies a set of events that have occurred "so far". An event can occur only if all the events in its past have occurred. Two events that are in conflict can never both occur in the same stretch of behaviour. Before formalizing these notions it will be convenient to adopt the following notion.

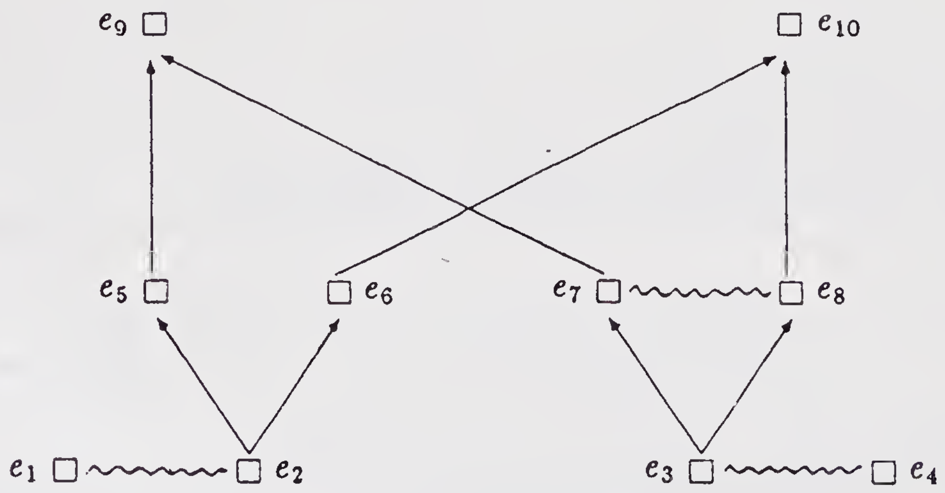


Figure 6. An event structure.

Let $ES=(E, \leqslant, \#)$ be an event structure and $X \subseteq E$. Then $\downarrow X = \{e' \mid \exists e \in X : e' \leqslant e\}$. For the singleton $\{e\}$, we shall write $\downarrow e$ instead of $\downarrow\{e\}$.

DEFINITION 3.2

Let $ES=(E, \leqslant, \#)$ be an event structure and $c \subseteq E$. Then c is a configuration iff

(1) $c = \downarrow c$, (left-closed)
(2) $(c \times c) \cap \# = \emptyset$. (conflict-free)

For the event structure shown in figure 6, $\{e_2, e_5, e_6\}$ is a configuration. $\{e_2, e_5, e_{10}\}$ is not a configuration because it is not left-closed and $\{e_3, e_7, e_8\}$ is not a configuration because it is not conflict-free.

We are particularly interested in a restricted subset of configurations called local configurations. The notion of a local configuration is based on a simple but crucial observation which lies at the heart of the theory of event structures (Nielsen *et al* 1980).

PROPOSITION 3.3.

Let $ES=(E, \leqslant, \#)$ be an event structure and $e \in E$. Then $\downarrow e$ is a configuration.

We now define $LC_{ES} = \{\downarrow e \mid e \in E\}$ to be the set of *local configurations* of the event structure $ES=(E, \leqslant, \#)$.

We do so because a (general) configuration $c \subseteq E$ can be viewed as a global state of the system. Part of a global configuration may change independent of each other, due to the spatial separation and the partial autonomy of the individual agents in the system being modelled by the event structure. A finite global configuration c is completely characterized by specifying the maximal events (with respect to $\leqslant$) which belong to c. Each local configuration $\downarrow e$ corresponding to a maximal event $e \in c$ can be regarded as a local state which contributes to the global state at c.

When we reason about the behaviour of an event structure, we would like to make assertions about properties that are satisfied by the global configurations – that is, properties that hold at the global states of the system. However, a global state can be *completely* described in terms of all the local states that are part of that global state. Thus, we shall restrict ourselves to specifying properties at the local configurations. Using combinations of these assertions, we can describe global configurations of the event structure. Further, the assertions that we can make about a global configuration are tied down to the assertions that we can make about the local configurations that constitute the global configuration. This will become clearer in the second part of the paper where we discuss how to specify properties of distributed systems.

As we had mentioned at the beginning of this section, an event structure is a single entity which describes all the computations of a distributed system. Thus, we need a means of "extracting" individual computations from an event structure. Since a configuration represents a set of events that have happened so far, in general, an arbitrary configuration represents a partial computation of the system. If we consider configurations which are maximal (with respect to inclusion) we obtain the maximal computations of the event structure. We call these the *runs* of the event structure. It is easy to verify the following characterization of runs. Let $r \subseteq E$. Then r is a run iff

$$\forall e \in E\text{: } e \in r \text{ iff } \forall e' \in E\text{: } e \# e' \text{ implies } e' \notin r.$$

Next, let us look at some useful restrictions on event structures. We begin with the auxiliary relation $\#_\mu$. In general, there may be events in # whose inconsistency cannot be traced back to a pair of events in $\#_\mu$ – a typical example consists of two infinite descending chains of events in # with each other. We would like to rule out such structures, since they model behaviours which are intuitively infeasible. We can therefore restrict our attention to well branching event structures.

DEFINITION 3.4

Let $ES = (E, \leqslant, \#)$ be an event structure. ES is *well-branching* iff

$$\forall e, e' \in E: e \# e' \text{ implies } \exists e_1, e'_1 \in E: e_1 \leqslant e \text{ and } e'_1 \leqslant e' \text{ and } e_1 \#_\mu e'_1.$$

Well-branching is a fairly weak restriction. A stronger and more useful restriction is that of *finitariness*. An event structure $ES = (E, \leqslant, \#)$ is said to be finitary in case $\downarrow e$ is a finite set for every $e \in E$. Finitariness captures the important fact that in any realizable system, an event can be causally dependent on only a finite set of events. An event with an infinite past can never actually occur.

There is a systematic way of describing the behaviour of elementary net systems using finitary event structures. To do this, we require *labelled* event structures. A labelled event structure is a pair $ES_\Sigma = (ES, \phi)$ where $ES = (E, \leqslant, \#)$ is an event structure and $\phi: E \to \Sigma$ is a labelling function.

Constructing a labelled finitary event structure describing the behaviour of a net system involves an intermediate stage where the net system is "unfolded" to generate an acyclic structure. The details are a bit involved and can be found in Nielsen *et al* (1980) and Thiagarajan (1990). We shall merely present an example.

Consider the elementary net system in figure 5 modelling mutual exclusion. The labelled event structure in figure 7 describes the behaviour of this system. In this case, the event occurrences in the event structure are labelled by the events of the net system.

Given a finitary event structure ES, we can construct a DTS DTS_{ES} which exhibits the same behaviour as ES. Let $\mathscr{C}_{ES}^{fin}$ denote the set of finite configurations of the

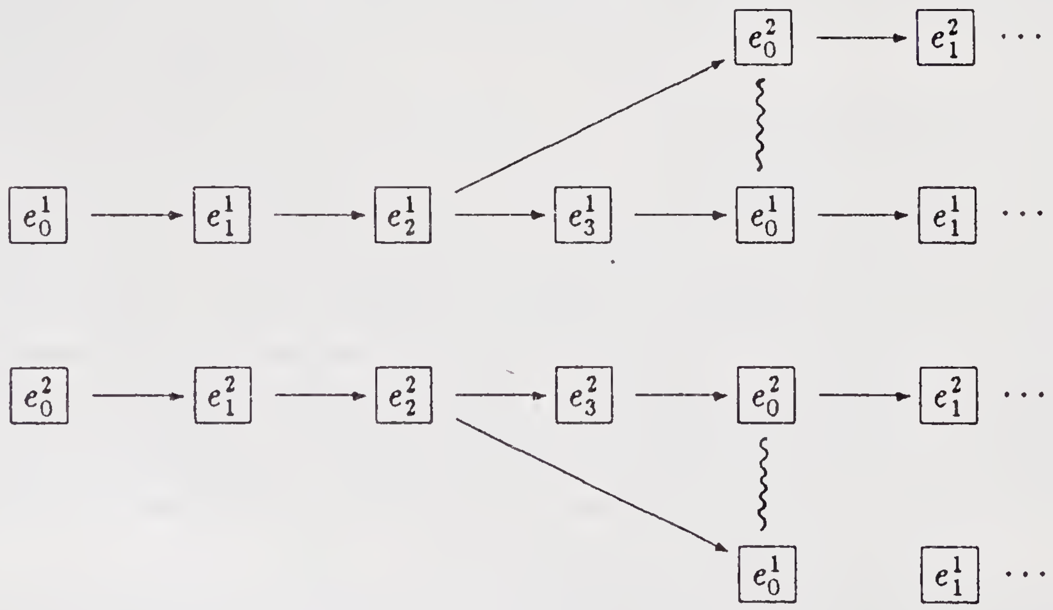


Figure 7. A labelled event structure.

finitary event structure $ES=(E,\leqslant,\#)$. We can define the step transition relation $\rightarrow_{ES}\subseteq\mathscr{C}^{fin}_{ES}\times\wp_{fin}(E)\times\mathscr{C}^{fin}_{ES}$ as follows:

$$\rightarrow_{ES}=\{(c,u,c')\mid c\cap u=\emptyset \text{ and } c\cup u=c' \text{ and }$$
$$\forall e_1,e_2\in u{:}\, e_1\neq e_2 \text{ implies } e_1\ co\ e_2\}.$$

PROPOSITION 3.5.

$DTS_{ES}=(\mathscr{C}^{fin}_{ES},E,\rightarrow_{ES})$ is a DTS *over E.*

As in the case of elementary net systems, it turns out that DTS_{ES} is always deterministic. Once again, we can use labelled event structures to permit non-determinism in this DTS. As before, we have to restrict the labelling to be *co-injective* to rule out multisets in concurrent steps. In other words, given $ES_\Sigma=(E,\leqslant,\#,\phi)$, we require that for every $e_1,e_2\in E$: $e_1 co e_2$ implies $\phi(e_1)\neq\phi(e_2)$. We then have the following result.

PROPOSITION 3.6.

Let $ES_\Sigma=(ES,\phi)$ be a Σ-labelled event structure where ϕ is a conjective labelling function. Then $DTS_{ES_\Sigma}=(\mathscr{C}^{fin}_{ES},\Sigma,\Rightarrow_{ES_\Sigma})$ is a DTS *over Σ where*

$$\Rightarrow_{ES_\Sigma}=\{(c,\phi(u),c')\mid(c,u,c')\in\rightarrow_{ES}\}.$$

2.4 *Communicating sequential agents*

In an event structure, the entire behaviour of a distributed system is specified as a single entity. Individual computations of the system can be identified using the notion of a run. However, no further information is provided about the structure of the system.

Consider a distributed system consisting of a finite set of sequential agents performing a joint task, using communication to coordinate their activities. When reasoning about the behaviour of such a system, it is convenient to associate the events occurring in the system with the agents involved in the events. This can be captured by restricting event structures to a model called communicating sequential agents (CSA).

Let **N** denote the set of natural numbers $\{1,2,3\ldots\}$. We shall use elements of **N** as names for the agents in our system.

DEFINITION 4.1

A system of *communicating sequential agents* (CSA) is a triple $CSA=(E,\leqslant,\eta)$, where

(1) E is a non-empty set of event occurrences;
(2) $\leqslant$ is a partial order on E called the causality relation;
(3) $\eta{:}E\rightarrow\wp_{fin}(\mathbf{N})$ is a naming function assigning to each e in E a non-empty finite subset of **N**;
(4) Let $E_j=\{e\mid e\in E \text{ and } j\in\eta(e)\}$. Then, for every e in E:

$$\forall j\in\mathbf{N}{:}\downarrow e\cap E_j \text{ is totally ordered by } \leqslant.$$

We interpret $j\in\eta(e)$ as the agent j participating in the event e. Thus $\eta(e)=\{1,2\}$ can stand for a synchronization "handshake" between agents 1 and 2.

The poset $(E_j, \leqslant_j)$, where $\leqslant_j$ is $\leqslant$ restricted to $E_j \times E_j$, represents the local behaviour of agent j in CSA. Usually, we say "agent j" to denote this poset.

As in an event structure, if $e_1 \leqslant e_2$ then e_2 causally depends on e_1; in no run of CSA can e_2 occur without e_1 having occurred earlier.

To separate concurrency from conflict, both the causality relation $\leqslant$ and the naming function η are used. In a CSA, each agent is defined to be sequential. Thus, given any two events e and e' which both involve the same agent – that is $\eta(e)$ and $\eta(e')$ are *not* disjoint – e and e' must either be causally related or in conflict. So if e and e' are incomparable with respect to $\leqslant$ and $\eta(e) \cap \eta(e') \neq \emptyset$, then e and e' are in conflict.

The motivation for the last condition in definition 4.1 should now be clear: we do not wish an event occurrence to causally depend upon conflicting event occurrences. This condition also implicitly ensures that the basic conflict in the system is generated within agents – in effect, choices are made locally by individual agents and then propagated across agents via $\leqslant$.

On the other hand, if two events e and e' are unordered and their combined past does not contain any conflicting events they must be concurrent. Since choices are assumed to be made locally, it is sufficient to check that for each agent j, the combined past of e and e' does not have incomparable events involving j. In other words, if $(\downarrow e \cup \downarrow e') \cap E_j$ is totally ordered by $\leqslant$ for every j, then the two events e and e' are concurrent.

If $e \in E_j$, the local state $\downarrow e$ includes the local history of agent j as well as the "latest" local histories of all other agents with which j has communicated upto this state. Let $LC_{CSA} = \{\downarrow e \mid e \in E\}$ be the set of local states of CSA.

By suitably restricting the naming function η, we can capture interesting subclasses of communicating sequential agents.

The first restriction is on the number of agents. In a general CSA, we may have an unbounded number of agents in the system. By restricting the range of η to a finite subset $\{1, 2, \ldots, n\}$ of $\mathbf{N}$, we obtain CSA which may have upto n agents, which we call n-CSA.

As we had mentioned earlier, if $\eta(e)$ is not a singleton, the interpretation is that the event e is performed jointly by the agents mentioned by $\eta(e)$. This intuitively corresponds to "handshaking" or synchronous communication between agents. By restricting η so that $|\eta(e)| = 1$ for every e in E, we effectively rule out this type of synchronous communication. Instead, in such an *asynchronous* CSA, the agents communicate by sending messages to each other. The sending and receiving of a message are regarded as two distinct actions, each involving only one agent at a time.

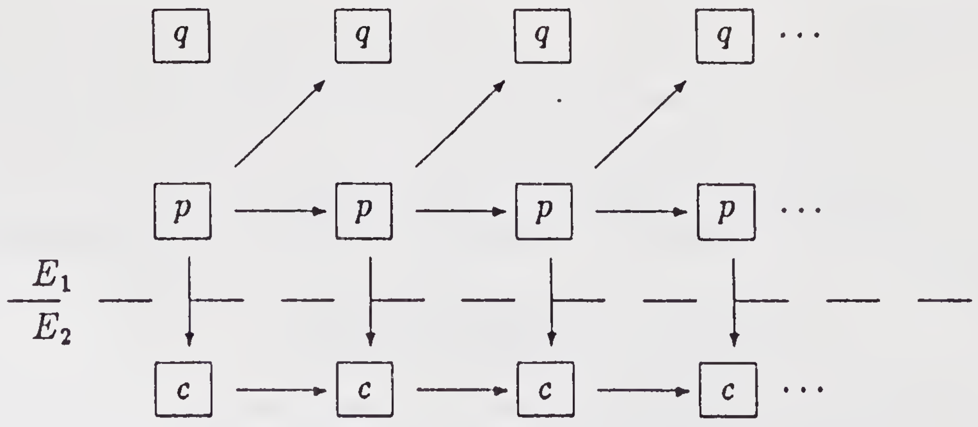


Figure 8. An asynchronous CSA.

Finally, we say that a CSA is *finitary* is case $\downarrow e$ is a finite set for every e in E. The motivation for defining finitary CSA is the same as the motivation for defining finitary event structures – any computation of a real system can be traced back to some starting point, so the past of any event occurring during the computation must be finite.

Figure 8 is an example of an asynchronous CSA consisting of two agents, a *producer* and a *consumer*, communicating via an unbounded buffer. The producer can produce zero or more items and then quit. The consumer can consume items produced by the producer as long as the items are available in the buffer. The events in the CSA are labelled p, q and c to denote these three types of actions.

3. Logics for concurrency

We now turn our attention to the problem of reasoning about the behaviour of distributed systems.

A *specification language* is simply a formalism in which one specifies behaviours of systems under study. Thus, a specification language for distributed systems is one in which we can describe behavioural properties of distributed systems.

The specification language should permit us to combine simple specifications together to construct more complex specifications, reflecting the intuition that large systems can be broken down into more manageable subsystems. This calls for disjunctive and conjunctive abilities in the language.

In addition, since we are dealing with distributed systems we expect to describe properties like causality, choice and concurrency. For this, we will need to be able to specify the relationships that hold between system states as the computation proceeds.

Our requirements suggest the use of a formal logic with boolean connectives and temporal modalities as our specification language. Temporal logic is a branch of modal logic which is used to study structures of states varying with time. We will design a variety of modal logics which are extensions of temporal logic to deal with the models of distributed systems developed in § 2.

We begin with a quick sketch of classical propositional modal logic. We assume the existence of $\mathscr{P}$, a countable set of atomic propositions $\{p_0, p_1, \ldots\}$. The well-formed formulas of our logic $\mathscr{L}_0$ are defined inductively:

- every $p \in \mathscr{P}$ is a formula of $\mathscr{L}_0$;
- if α and β are formulas of $\mathscr{L}_0$, then so are $\neg\alpha, \alpha \vee \beta$ and $\Diamond\alpha$.

($\neg\alpha$ is to be read as "not α", $\alpha \vee \beta$ is to be read as "α or β", and $\Diamond\alpha$ is to be read as "diamond α"). The intended meaning of $\Diamond\alpha$ is "α becomes true eventually".

Formulas are to be interpreted over *frames*. In our set-up, a *frame* is a transition system $TS = (S, \Sigma, \rightarrow)$. A *model* M is a frame with a *valuation* function; i.e. $M = (TS, V)$, where $TS = (S, \Sigma, \rightarrow)$ is a transition system and $V: S \rightarrow \wp(\mathscr{P})$. For example, if $V(s) = \{p_1, p_3\}$, we interpret this to mean that propositions p_1 and p_3 are true at state s and, further, that no other proposition is true at s.

The notion of a formula α being true at a state s in a model $M = (TS, V)$ where $TS = (S, \Sigma, \rightarrow)$, denoted as $M, s \models \alpha$, is defined inductively as follows:

(i) $M, s \models p$ iff $p \in V(s)$, for $p \in \mathscr{P}$;
(ii) $M, s \models \neg\alpha$ iff $M, s \not\models \alpha$;
(the notation $M, s \not\models \alpha$ stands for "it is not the case that $M, s \models \alpha$");

(iii) $M, s \models \alpha \vee \beta$ iff $M, s \models \alpha$ or $M, s \models \beta$.
(iv) $M, s \models \Diamond\alpha$ iff $\exists s' \in \mathscr{R}(s)$: $M, s' \models \alpha$;
(recall that $\mathscr{R}(s)$ is the set of states reachable from s via $\rightarrow$).

$M, s \models \alpha$ can be interpreted as the assertion that the model M at state s is an implementation of the specification α. We say α is *satisfiable* if there exists a model $M = (TS, V)$, where $TS = (S, \Sigma, \rightarrow)$, and there exists a state $s \in S$ such that $M, s \models \alpha$. We say that α is M-valid if $M, s \models \alpha$ for every $s \in S$. We say that α is valid – and denote this by $\models \alpha$ – if α is M-valid for every model M. It is easy to see that α is valid iff $\neg\alpha$ is not satisfiable.

The following derived formulas are useful.

$\alpha \wedge \beta \overset{\text{def}}{=} \neg(\neg\alpha \vee \neg\beta)$,	the conjunction of α and β,
$\alpha \supset \beta \overset{\text{def}}{=} \neg\alpha \vee \beta$,	α implies β,
$\alpha \equiv \beta \overset{\text{def}}{=} (\alpha \supset \beta) \wedge (\beta \supset \alpha)$,	logical equivalence of α and β,
$\Box\alpha \overset{\text{def}}{=} \neg(\Diamond\neg\alpha)$,	"Henceforth" α,
$\mathit{True} \overset{\text{def}}{=} p_0 \vee \neg p_0$,	
$\mathit{False} \overset{\text{def}}{=} \neg\,\mathit{True}$.	

It can easily be verified that for any model $M = ((S, \rightarrow), V)$ and $s \in S$,

$$M, s \models \Box\alpha \text{ iff } \forall s' \in \mathscr{R}(s): M, s' \models \alpha.$$

A number of interesting properties of transition systems can be expressed using this logic. Suppose that we are using transition systems to model a distributed system consisting of n processes which can compete for a shared resource r. Let the atomic proposition c_i stand for " Process i has access to the resource r". Then,

$$\Box \bigwedge_{i \in \{1,2,\ldots,n\}} \left(c_i \supset \bigwedge_{i \neq j} \neg c_j \right),$$

expresses a so-called *safety property*. It says that at any system state, at most one process has control of the shared resource r. This will ensure, for instance, that in case r is a shared piece of data then the sequence of values assumed by r during the history of the system will be well-defined. Broadly speaking, safety properties assert that "bad" situations never arise in the system.

Similarly, if we let the proposition rq_i stand for "Process i requires access to resource r", the formula,

$$\Box \bigwedge_{i \in \{1,2,\ldots,n\}} (rq_i \supset \Diamond c_i),$$

expresses a *liveness property*. It says that any request made by a process for the shared resource is eventually granted by the system. In general, liveness properties specify that something "good" occurs eventually.

This logical framework is very simple, but for that reason is also not as expressive as we would wish. In particular, we would like to devise logics to reason about models with true concurrency. In the rest of this section, we shall show how such logics can be defined for the formal models presented in § 2.

3.1 *Logic for distributed transition systems*

Recall that in a DTS, a concurrent step consists of a transition labelled by a finite set of actions. This leads us to augment the simple modal logic considered earlier with one additional modality, $\langle u \rangle$, where u is a finite subset of Σ, the set of actions.

Let $\mathscr{L}_{DTS}$ be the language whose well-formed formulas are given by:

- every $p \in \mathscr{P}$ is a formula of $\mathscr{L}_{DTS}$;
- if α and β are formulas of $\mathscr{L}_{DTS}$ then so are $\neg \alpha, \alpha \vee \beta$, $\Diamond \alpha$ and $\langle u \rangle \alpha$, where u is a finite subset of Σ.

Thus, the logic $\mathscr{L}_{DTS}$ is parametrized by Σ. To emphasize this, we will write $\mathscr{L}^{\Sigma}_{DTS}$. instead of $\mathscr{L}_{DTS}$.

As one may expect, the frames for our logic are distributed transition systems over Σ. A model is a pair $M = (DTS, V)$, where $DTS = (S, \Sigma, \rightarrow)$ is a DTS over Σ and $V: S \rightarrow \wp(\mathscr{P})$ is the valuation function. Given $s \in S$, the notion $M, s \models \alpha$ is defined as before for the atomic propositions and for the connectives $\neg$ and $\vee$ and the modality $\Diamond$. For the new modality we define:

$$M, s \models \langle u \rangle \alpha \text{ iff } \exists s' \in S : s \xrightarrow{u} s' \text{ and } M, s' \models \alpha.$$

Relative to the new notion of models, satisfiability and validity are defined as before. We will write $\models^{\Sigma}_{DTS} \alpha$ to denote that α is a valid formula in this logic. Let SAT^{Σ}_{DTS} denote the set of all satisfiable formulas from $\mathscr{L}^{\Sigma}_{DTS}$.

Before considering an example, we introduce some notational conventions. The derived modality $[u]$ is defined as:

$$[u]\alpha \overset{\text{def}}{=} \neg \langle u \rangle \neg \alpha.$$

where u is a singleton $\{a\}$, we will write $\langle a \rangle \alpha$ instead of $\langle \{a\} \rangle \alpha$. For the empty step, we write $\langle \emptyset \rangle \alpha$.

Now that the modalities are indexed by steps, we can clearly identify the branching points in a transition system. For example, consider the transition systems shown in figure 9. In the first system, starting at s_0 we can perform a and then choose between b and c whereas in the second system, at s'_0 we have to decide right away whether we are going to execute a followed by b or a followed by c. The first situation is captured by the formula $\langle a \rangle(\langle b \rangle\, True \wedge \langle c \rangle\, True)$ while the second can be expressed as $\langle a \rangle(\langle b \rangle\, True \wedge [c]\, False) \wedge \langle a \rangle(\langle c \rangle\, True \wedge [b]\, False)$.

In this logic, we can distinguish between interleavings and true concurrency. For instance, the formula $\langle a \rangle \langle b \rangle\, True \wedge \langle b \rangle \langle a \rangle\, True \wedge [\{a, b\}]\, False$ is satisfiable. At the state where this formula is true, both the interleavings ab and ba can occur, but

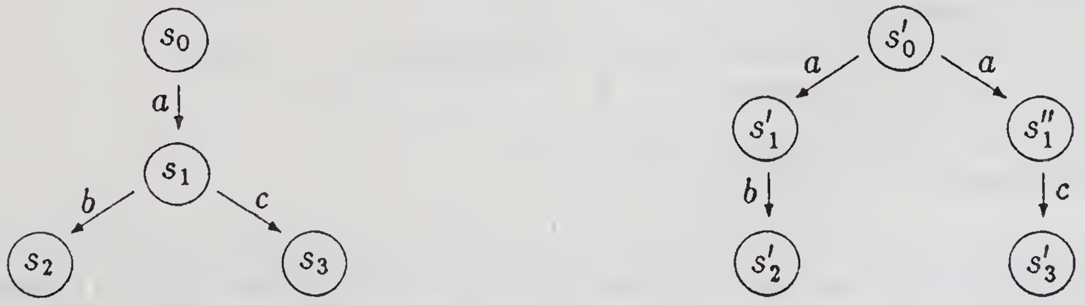


Figure 9. Varieties of branching in transition systems.

the corresponding concurrent step $\{a, b\}$ is not enabled. On the other hand, it is easy to see that the formula $\langle\{a, b\}\rangle\alpha \supset \langle a\rangle\langle b\rangle\alpha$ is a valid formula, because the definition of a DTS guarantees the existence of a function f associated with each step, breaking it up into substeps.

Returning briefly to the sytem of n processes considered earlier, assume that the shared resource r represents a data item in a shared block of memory. Let ud_i denote the act of process i updating the value of r. Then, the specification

$$\Box \bigwedge_{i \neq j} [\{ud_i, ud_j\}]False,$$

requires that the memory manager never permit two distinct processes to concurrently update r.

Let us consider another example. The writing of a paper can be seen as a sequential activity: work out what you want to say, write it out, get it typed. In the case of a joint paper, the work may be divided up in terms of sections. One policy the authors may follow is to work out all the sections before preparing a typescript, with meetings for discussion and correction in between. That is, the authors satisfy

$$\langle WK\rangle(worked \wedge \langle WR\rangle(written \wedge \langle TY\rangle typed)),$$

where

$$WK = \{\text{work out §1, work out §2, work out §3}\},$$

$$WR = \{\text{write §1, write §2, write §3}\},$$

$$TY = \{\text{type §1, type §2, type §3}\},$$

and *worked*, *written* and *typed* are atomic propositions indicating the end of the working out, written and typing steps respectively. Here we have assumed that there are three authors each of whom is responsible for one section.

The concurrent steps are necessary, since they express the fact that this is a *joint* paper; if the interleaving of the actions required for the three sections were present, we could not rule out the possibility that the three authors were separately writing three (single-section) papers.

The states we are using are global states. The person working out §2 may refer to a lemma in §1; the person doing the word processing for §1 may use the macros defined in §3.

It becomes necessary to use sequentializations when a complete record of the writing of the paper is required. For example, a mistake pointed out by the referee in §2 may be traced to the lemma in §1, which may be just a case of wrong typing thanks to a misapplication of the macro from §3.

This sort of mixture of independent actions and synchronization is well described in a DTS framework.

We now turn to the formal theory of the language $\mathscr{L}^{\Sigma}_{DTS}$. Typical questions one asks of such a logic include:

- Is the set of valid formulas axiomatizable?
- Is the satisfiability problem decidable?

The answers to these questions provide a good deal of insight into the strengths and weaknesses of the logic and, most importantly, into the expressive power of the logic.

It turns out that both these questions have positive answers for $\mathscr{L}^{\Sigma}_{DTS}$. Consider the following logical system ND.

The system ND

AXIOM SCHEMES

(A0) All the substitutional instances of the tautologies of Propositional Calculus.

(A1) (a) $\Box(\alpha \supset \beta) \supset (\Box\alpha \supset \Box\beta)$ (Deductive Closure)

(b) $[u](\alpha \supset \beta) \supset ([u]\alpha \supset [u]\beta)$

(A2) $\Box\alpha \supset [u]\alpha \wedge \Box\Box\alpha$ (Reachability)

(A3) $\alpha \equiv \langle\emptyset\rangle\alpha$ (Empty Step)

(A4,k) (for $k \geqslant 1$) (Step Axiom)

$$\langle u\rangle\alpha \wedge \bigwedge_{v \subseteq u} [v] \bigvee_{i=1}^{k} \beta_v^i \supset \bigvee_{f \in F(u,k)} \bigwedge_{v_1 \subseteq u} \langle v_1\rangle \left(\gamma_{v_1} \wedge \bigwedge_{v_1 \subseteq v_2 \subseteq u} \langle v_2 - v_1\rangle \gamma_{v_2} \right)$$

where $F(u,k)$ is the set of all functions $\{f \mid f: \wp(u) \to \{1,2,\ldots,k\}\}$ and

$$\gamma_v = \begin{cases} \beta_v^{f(v)} \wedge \alpha, & \text{if } v = u, \\ \beta_v^{f(v)}, & \text{if } v \subset u. \end{cases}$$

INFERENCE RULES

$$(\text{MP})\frac{\alpha, \alpha \supset \beta}{\beta} \quad (\text{TG})\frac{\alpha}{\Box\alpha}.$$

Axioms A0 to A2 and the rules MP and TG are standard. The characteristic axioms of distributed transition systems are A3 and A4,k. A3 captures the fact that the empty step cannot change the state of the system. A4,k is actually an infinite set of axioms, finitely presented. The complicated formulation of A4,k is necessary to describe the fact that each concurrent step u in a DTS can be broken up into concurrent substeps which are specified by the associated function $f: \wp(u) \to S$.

A formula α is called a *thesis* of the system ND – denoted $\vdash_{ND}\alpha$ – iff α can be derived in a finite number of steps using the axioms and inference rules of ND.

Theorem 1.1. (1) *ND is a sound and complete axiomatization of the valid formulas in* $\mathscr{L}_{DTS}^{\Sigma}$. *In other words,* $\vdash_{ND}\alpha$ *iff* $\models_{DTS}^{\Sigma}\alpha$ *for every* $\alpha \in \mathscr{L}_{DTS}^{\Sigma}$.
(2) *The satisfiability problem for this logic* (*i.e. the membership problem for* SAT_{DTS}^{Σ}) *is decidable in nondeterministic exponential time.*

It turns out that combining concurrency, captured by the step notion, with determinacy leads to a very expressive class of models. The frame $TS = (S, \Sigma, \to)$ is said to be deterministic if for every $s \in S$ and every $u \in \wp_{fin}(\Sigma)$ there exists at most one $s' \in S$ such that $s \xrightarrow{u} s'$. A model is deterministic if its underlying frame is.

The formula α is said to be deterministically satisfiable if there exists a deterministic model for α. Similarly, α is said to be deterministically valid if α is valid over the class of deterministic models. Let $\models_{Det}^{\Sigma}\alpha$ denote that α is deterministically valid and let $DSAT_{DTS}^{\Sigma}$ denote the set of deterministically satisfiable formulas in $\mathscr{L}_{DTS}^{\Sigma}$.

It turns out that the deterministically valid formulas in $\mathscr{L}_{DTS}^{\Sigma}$ are axiomatizable. Thanks to determinacy, one obtains a much simpler axiomatization than for the general case. Let D denote the logical system obtained from ND by dropping the infinitary set of axioms A4, $k (k \geqslant 1)$ and adding two new axioms:

(A5) $\langle u\rangle\alpha \supset \langle v\rangle\langle u - v\rangle\alpha$, $(v \subseteq u)$, (Weak Step Axiom)

(A6) $\langle u\rangle\alpha \supset [u]\alpha$. (Determinacy)

Let $\vdash_D\alpha$ denote that α is derivable in D.

Theorem 1.2. (1) *D is a sound and complete axiomatization of the deterministically valid formulas in $\mathscr{L}^{\Sigma}_{DTS}$. In other words, $\vdash_D \alpha$ iff $\models^{\Sigma}_{Det} \alpha$ for every $\alpha \in \mathscr{L}^{\Sigma}_{DTS}$.*
(2) *The membership problem for $DSAT^{\Sigma}_{DTS}$ is* undecidable.

The surprise here is that determinacy adds a sufficient amount of expressive power to make the satisfiability problem undecidable. By combining concurrent steps in a deterministic fashion, it turns out that we can encode the two-dimensional grid of natural numbers $\mathbf{N} \times \mathbf{N}$. We can then use this encoding to reduce some undecidable tiling problems described by Wang (1961) and Harel (1985) to the problem of deterministic satisfiability in our logic. This negative result was shown by Parikh (1989, pp. 199–209).

A variety of positive and negative results can be obtained in this logical framework by studying the effect of placing suitable restrictions on the DTS. For instance, we can restrict the set of actions Σ to be finite. Alternatively, we can demand that the DTS as a whole be finite – that is, the set of states and the set of transitions both be finite. We can also incorporate ideas from trace theory, arising out of the work of Mazurkiewicz (1989, pp. 285–363), and define trace transition systems, which permit both local and global specifications of concurrency. Finally, we can also study a smooth generalization of Propositional Dynamic Logic (Harel 1984, pp. 497–604) obtained by extending the notion of a regular program to permit concurrent steps as atomic actions. The details can be found in a forthcoming paper (Lodaya *et al* 1991).

The logical language $\mathscr{L}^{\Sigma}_{DTS}$ can also be interpreted over Σ-labelled elementary net systems and Σ-labelled event structures, where the labelling function is co-injective. The frames that we use are the corresponding DTS, as defined in § 2. Thus, a Σ-labelled elementary net system $\mathscr{N}_{\Sigma} = (\mathscr{N}, \phi)$, where $\mathscr{N} = (B, E, F, c_{in})$, gives rise to a model $(DTS_{\mathscr{N}_{\Sigma}}, V)$, where $V: C_{\mathscr{N}} \to \wp(\mathscr{P})$. Similarly, a Σ-labelled event structure $ES_{\Sigma} = (ES, \phi)$, where $ES = (E, \leqslant, \#)$, defines a model $(DTS_{ES_{\Sigma}}, V)$, where $V: \mathscr{C}^{fin}_{ES} \to \wp(\mathscr{P})$.

Let $SAT^{\Sigma}_{\mathscr{N}}$ and SAT^{Σ}_{ES} denote the set of formulas from $\mathscr{L}^{\Sigma}_{DTS}$ satisfiable in models generated by Σ-labelled elementary net systems and Σ-labelled event structures respectively.

Theorem 1.3. $SAT^{\Sigma}_{DTS} = SAT^{\Sigma}_{\mathscr{N}} = SAT^{\Sigma}_{ES}$.
In other words, this logic cannot discriminate between these classes of models.

3.2 *Logic for event structures*

We now turn from distributed transition systems to event structures as frames for our logic. In the logic for DTS, we used the global state approach to reasoning about the behaviour of the system. In this approach, assertions are made by a "global" observer of the system who can "see" the distributed system in its entirety in any given state. This is appropriate for DTS since the states of a DTS do in fact correspond to the global states of the system being modelled.

Alternatively, we can reason about the system from the point of view of the local states of the system. Here, assertions are made by individual agents in the system and hence the nature of the assertion is determined by the "visibility" of the system state from that agent's point of view. This approach is more suitable for reasoning based on event structures, where we can use a local configuration $\downarrow e$ to represent the local state of the system at the point where the event e has just occurred.

Another feature of the DTS logic is that concurrency is described by explicitly specifying the actions which are to be performed concurrently and describing the effect of such actions. This approach is natural for the DTS because the models themselves are action-based. On the other hand, in an event structure it is more convenient to specify concurrency in an abstract manner by simply asserting facts about concurrent events without specifying which actions are to be performed concurrently.

The key notions in the theory of event structures are those of causality, conflict and concurrency. This leads us to extend the language $\mathscr{L}_0$ by adding modalities to capture these notions. It turns out to be fruitful to split up causality into two parts, allowing us to specify both "past" and "future" behaviour.

The logic $\mathscr{L}_{ES}$ is built up as follows: again fix $\mathscr{P} = \{p_0, p_1, \dots\}$, a countable set of atomic propositions. Then the well-formed formulas of $\mathscr{L}_{ES}$ are given by:

- every $p \in \mathscr{P}$ is a formula of $\mathscr{L}_{ES}$;
- if α and β are formulas of $\mathscr{L}_{ES}$, then so are $\neg\alpha$, $\alpha \vee \beta$, $\Diamond\alpha$, $\diamondminus\alpha$, $\triangle\alpha$ and $\nabla\alpha$.

Here, the modalities $\Diamond$ and $\diamondminus$ denote the future and past respectively. $\triangle$ will be used to describe concurrency and ∇ will be used to capture conflict.

Frames for this logic are event structures, or rather the local configurations of event structures. More precisely, a frame is a pair (ES, LC_{ES}), where $ES = (E, \leqslant, \#)$ is an event structure and LC_{ES} is the set of local configurations of ES.

A model is a pair $M = ((ES, LC_{ES}), V)$ where ES is a frame and $V: LC_{ES} \to \wp(\mathscr{P})$ is a valuation function. If $p \in V(\downarrow e)$ then this is taken to mean that p is true at the local state $\downarrow e$ in the model M.

The notion of a formula α being true at a local state $\downarrow e$ in the model $M = ((ES, LC_{ES}), V)$ is denoted as $M, \downarrow e \models \alpha$ and is defined inductively as follows.

(i) $M, \downarrow e \models p$, iff $p \in V(\downarrow e)$, for $p \in \mathscr{P}$.
(ii) $M, \downarrow e \models \neg\alpha$, iff $M, \downarrow e \not\models \alpha$.
(iii) $M, \downarrow e \models \alpha \vee \beta$, iff $M, \downarrow e \models \alpha$ or $M, \downarrow e \models \beta$.
(iv) $M, \downarrow e \models \Diamond\alpha$, iff $\exists e' : e < e'$ and $M, \downarrow e' \models \alpha$.
(v) $M, \downarrow e \models \diamondminus\alpha$, iff $\exists e' : e' < e$ and $M, \downarrow e' \models \alpha$.
(vi) $M, \downarrow e \models \nabla\alpha$, iff $\exists e' : e \# e'$ and $M, \downarrow e' \models \alpha$.
(vii) $M, \downarrow e \models \triangle\alpha$, iff $\exists e' : e\, co\, e'$ and $M, \downarrow e' \models \alpha$.

Notice that we have defined the modalities $\Diamond$ and $\diamondminus$ in an *irreflexive* manner. This is necessary for the axiomatization which follows.

The notions of satisfiability and validity are defined as usual. $\models_{ES} \alpha$ will denote that α is a valid formula in $\mathscr{L}_{ES}$.

The derived connectives $\wedge$, $\supset$, $\equiv$, $\Box$ are defined as before. In addition, we set

$$\boxminus\alpha \overset{\text{def}}{=} \neg\diamondminus\neg\alpha, \quad \dot{\nabla}\alpha \overset{\text{def}}{=} \neg\nabla\neg\alpha, \quad \dot{\triangle}\alpha \overset{\text{def}}{=} \neg\triangle\neg\alpha.$$

We can also define a useful derived modality as follows:

$$\mathscr{S}\alpha \overset{\text{def}}{=} \alpha \vee \Diamond\alpha \vee \diamondminus\alpha \vee \nabla\alpha \vee \triangle\alpha.$$

$\mathscr{S}\alpha$ is to be read as "Somewhere α". Its dual $\mathscr{E}\alpha \overset{\text{def}}{=} \neg\mathscr{S}\neg\alpha$, read as "Everywhere α", expands as follows:

$$\mathscr{E}\alpha \overset{\text{def}}{=} \alpha \wedge \Box\alpha \wedge \boxminus\alpha \wedge \dot{\nabla}\alpha \wedge \dot{\triangle}\alpha.$$

Thus $\mathscr{E}\alpha$ describes a property invariant over the entire model.

Many interesting features of event structures can be expressed in this logic. Recall that the maximal computations of event structures are termed runs. We can use an atomic proposition ρ to mark out a run with the formula $\rho \equiv \dot{\triangledown} \neg \rho$. For any modal $M = ((ES, LC_{ES}), V)$ if the formula $\rho \equiv \dot{\triangledown} \neg \rho$ is M-valid, then $\{e | M, \downarrow e \models \rho\}$ constitutes a run of ES. Using this method of marking out runs, we can express liveness and safety properties in event structures. Let α represent a liveness property. Then $\mathscr{S}(\rho \wedge \alpha)$ is M-valid for a model M just in case every computation of the underlying event structure contains a local state where α is true. Similarly, if β represents an undesirable situation, the formula $\mathscr{E}(\rho \supset \neg \beta)$ expresses the safety property that β does occur at any state of the run marked by ρ.

In a similar spirit the formula $\chi \equiv \Box \neg \chi \wedge \boxminus \neg \chi$ can be used to capture the notion of a cut – a maximal set of pair-wise incomparable events. Within a computation, a cut corresponds to a global state. Thus we can use the notion of a cut in conjunction with that of a run to look "sideways" from a local state and make assertions about the current global state.

The formula $\triangledown \alpha \supset \Box \triangledown \alpha$ describes the fact that conflict is inherited in a prime event structure. The formula $\triangle \alpha \supset \boxminus (\triangle \alpha \vee \Diamond \alpha)$ expresses the fact that the configurations of an event structure are "consistent" by asserting that the unified past of any pair of events in *co* is conflict-free.

Due to lack of space, we will not provide a separate detailed example for this logic. The logic presented in the next section, called $\mathscr{L}_{CSA}$, is also based on event structures. We shall provide a detailed example for that logic. It will not be difficult to see how that example can be translated into the present framework.

Consider the logical system E.

The system E

AXIOM SCHEMES

(A0) All the substitutional instances of the tautologies of Propositional Calculus.

(A1) (i) $\Box(\alpha \supset \beta) \supset (\Box \alpha \supset \Box \beta)$ (Deductive closure)

(ii) $\boxminus(\alpha \supset \beta) \supset (\boxminus \alpha \supset \boxminus \beta)$

(iii) $\dot{\triangledown}(\alpha \supset \beta) \supset (\dot{\triangledown} \alpha \supset \dot{\triangledown} \beta)$

(iv) $\dot{\triangle}(\alpha \supset \beta) \supset (\dot{\triangle} \alpha \supset \dot{\triangle} \beta)$

(A2) (i) $\Box \alpha \supset \Box \Box \alpha$ (Transitivity of $<$)

(ii) $\boxminus \alpha \supset \boxminus \boxminus \alpha$

(A3) (i) $\alpha \supset \dot{\triangledown} \triangledown \alpha$ (Symmetry of # and *co*)

(ii) $\alpha \supset \dot{\triangle} \triangle \alpha$

(A4) (i) $\alpha \supset \Box \diamondminus \alpha$ (Relating past and future)

(ii) $\alpha \supset \boxminus \Diamond \alpha$

(A5) $\triangledown \alpha \supset \Box \triangledown \alpha$ (Conflict inheritance)

(A6) $\triangle \alpha \supset \boxminus (\Diamond \alpha \vee \triangle \alpha)$ (Conflict-free past)

(A7) (i) $\Diamond \alpha \supset \Box (\alpha \vee \Diamond \alpha \vee \diamondminus \alpha \vee \triangledown \alpha \vee \triangle \alpha)$ (Relating $<$, # and *co*)

(ii) $\triangledown \alpha \supset \dot{\triangledown} (\alpha \vee \Diamond \alpha \vee \diamondminus \alpha \vee \triangledown \alpha \vee \triangle \alpha)$

(iii) $\triangle \alpha \supset \dot{\triangle} (\alpha \vee \Diamond \alpha \vee \diamondminus \alpha \vee \triangledown \alpha \vee \triangle \alpha)$

(iv) $\diamondminus \alpha \supset \boxminus (\alpha \vee \Diamond \alpha \vee \diamondminus \alpha \vee \triangle \alpha)$

(v) $\triangledown \alpha \supset \dot{\triangle} (\Diamond \alpha \vee \triangledown \alpha \vee \triangle \alpha)$

(vi) $\triangle \alpha \supset \Box (\diamondminus \alpha \vee \triangledown \alpha \vee \triangle \alpha)$

INFERENCE RULES

(MP) $\dfrac{\alpha}{\alpha \supset \beta}$

(TG)(i) $\dfrac{\alpha}{\Box\alpha}$ (ii) $\dfrac{\alpha}{\boxminus\alpha}$ (iii) $\dfrac{\alpha}{\dot{\triangle}\alpha}$ (iv) $\dfrac{\alpha}{\dot{\nabla}\alpha}$

(UNIQ) $\dfrac{\hat{p} \supset \alpha}{\alpha}$, where p is an atomic proposition not appearing in α and

$$\hat{p} \overset{\mathrm{def}}{=} p \wedge \Box \sim p \wedge \boxminus \sim p \wedge \dot{\triangle} \sim p \wedge \dot{\nabla} \sim p.$$

Axioms A0 to A4 and inference rules MP and TG are standard. A5 expresses the fact that conflict is inherited via $\leqslant$. A6 ensures that any two events related by *co* have consistent (i.e. conflict-free) pasts. The remaining axioms are necessary to capture the fact that the relations $\leqslant$, $\geqslant$, # and *co* "cover" the event structure – i.e. any two distinct events are related by one of these relations.

The rule UNIQ is adapted from Burgess (1980). Given a proposition p, the definition of $\hat{p}$ ensures that it can be true in at most one local configuration. Hence, we can label each local configuration $\downarrow e$ by a distinct formula $\hat{p}_e$. The rule UNIQ allows us to construct this labelling, which is crucial in demonstrating the completeness of the axiomatization.

Let $\vdash_E \alpha$ denote that α is a thesis of the system E.

Theorem 2.1. *E is a sound and complete axiomatization of the valid formulas in $\mathscr{L}_{ES}$. In other words, $\vdash_E\alpha$ iff $\models_{ES} \alpha$.*

Recall that we had defined an auxiliary relation $\#_\mu$ in an event structure, called the minimal conflict relation. We can define a modality ∇_μ to capture the relation $\#_\mu$.

It is possible to strengthen $\mathscr{L}_{ES}$ by replacing the modality ∇ by the modality ∇_μ. Let us call this new language $\mathscr{L}^\mu{}_{ES}$. To obtain a useful comparison with $\mathscr{L}_{ES}$, and also to obtain an axiomatization, we must change the notion of a frame. For this language, we define a frame to be a pair (ES, LC_{ES}) where ES is a *well-branching* event structure. Recall that a well-branching event structure is one in which the # relation can be completely specified using the relations $\#_\mu$ and $\leqslant$. As usual, a model is a frame together with a valuation function. Models based on well branching frames are called well branching models.

The semantics of $\mathscr{L}^\mu_{ES}$ is the same as that of $\mathscr{L}_{ES}$ except that the clause for ∇ is replaced by:

$$M, \downarrow e \models \nabla_\mu \alpha \text{ iff } \exists e' : e \#_\mu e' \text{ and } M, \downarrow e' \models \alpha.$$

In $\mathscr{L}^\mu_{ES}$, we can obtain ∇ as a derived modality:

$$\nabla\alpha \overset{\mathrm{def}}{=} \nabla_\mu\alpha \vee \nabla_\mu \Diamond\alpha \vee \diamondminus \nabla_\mu\alpha \vee \diamondminus \nabla_\mu \Diamond\alpha.$$

As before, $\dot{\nabla}\alpha$ denotes the formula $\neg \nabla \neg \alpha$. It is easy to verify that $\dot{\nabla}\alpha$ can be expressed as follows:

$$\dot{\nabla}\alpha \overset{\mathrm{def}}{=} \dot{\nabla}_\mu\alpha \wedge \dot{\nabla}_\mu \Box\alpha \wedge \boxminus \dot{\nabla}_\mu\alpha \wedge \boxminus \dot{\nabla}_\mu \Box\alpha.$$

In a well-branching model, the derived modalities $\triangledown$ and $\dot{\triangledown}$ have precisely the same interpretation as the corresponding modalities of $\mathscr{L}_{ES}$. On the other hand, there is no obvious way to characterize the minimal conflict relation $\#_\mu$ using the modality $\triangledown$. In this connection, we can establish the following result.

Theorem 2.2. *For well-branching models, the language $\mathscr{L}_{ES}$ is strictly more expressive than $\mathscr{L}_{ES}$.*

Informally, this result says that we can use formulas from $\mathscr{L}^{\mu}_{ES}$ to differentiate models which are indistinguishable using the language $\mathscr{L}_{ES}$.

An example of the use of $\triangledown_\mu$ is in systems where agents have names, like communicating sequential agents. For each event e that process i participates in, we can assign an atomic proposition τ_i to the local configuration $\downarrow e$. Suppose that there are n agents in the system, with "names" $\tau_1, \tau_2, \ldots, \tau_n$. Then the formula $\bigwedge_{1 \leqslant i \leqslant n} (\tau_i \supset \dot{\triangledown}_\mu \tau_i)$ expresses the fact that all choices in behaviour are made locally by individual agents.

The axiom system E_μ is obtained by adding the following axiom schemes to the system E.

(A1) (v) $\dot{\triangledown}_\mu(\alpha \supset \beta) \supset (\dot{\triangledown}_\mu \alpha \supset \dot{\triangledown}_\mu \beta)$, (Deductive Closure)
(A3) (iii) $\alpha \supset \dot{\triangledown}_\mu \triangledown_\mu \alpha$, (Symmetry of $\#_\mu$)
(A6) (ii) $\triangledown_\mu \alpha \supset \boxdot(\Diamond \alpha \vee \triangle \alpha)$, (Minimal Conflict)

A1(v) and A3(iii) are standard. A6(ii) is the characteristic axiom describing the $\#_\mu$ relation as the minimal conflict relation.

Let $\vdash^{\mu}_{E} \alpha$ denote that α is a thesis of the system E_μ and let $\models^{\mu}_{ES} \alpha$ denote that α is valid over the class of well branching models. Then we get:

Theorem 2.3. *E_μ is a sound and complete axiomatization of the valid formulas in $\mathscr{L}^{\mu}_{ES}$. In other words, $\vdash^{\mu}_{E} \alpha$ iff $\models^{\mu}_{ES} \alpha$.*

3.3 *Logic for communicating sequential agents*

We now wish to study a means of talking about a central feature of many distributed systems – the communication pattern between the components of the system that ensure coordination. For this, we shall define a logic that is to be interpreted over communicating sequential agents.

Let $\mathscr{P} = \{p_0, p_1, \ldots\}$ be a countable set of atomic propositions, and $\mathscr{T} = \{\tau_0, \tau_1, \ldots\}$, a countable set of type propositions disjoint from $\mathscr{P}$. The formulas of $\mathscr{L}_{CSA}$ are built up as follows:

- every member of $\mathscr{P} \cup \mathscr{T}$ is a formula of $\mathscr{L}_{CSA}$;
- if α and β are formulas of $\mathscr{L}_{CSA}$, then so are $\neg\alpha$, $\alpha \vee \beta$, $\Diamond_i \alpha$ and $\diamondminus_i \alpha$.

The formula τ_i asserts that the observer is located in agent i. $\Diamond_i$ and $\diamondminus_i$ capture the "visible" future and past of agent i. This will become clearer when we define the formal semantics of these modalities.

A frame for $\mathscr{L}_{CSA}$ is a pair (CSA, LC_{CSA}), where $CSA = (E, \leqslant, \eta)$ is a system of communicating sequential agents and LC_{CSA} is the set of local states of CSA. A model is a pair $M = ((CSA, LC_{CSA}), V)$ where (CSA, LC_{CSA}) is a frame and $V\colon LC_{CSA} \to \wp(\mathscr{P} \cup \mathscr{T})$ is a valuation function such that

$$\tau_i \in V(\downarrow e) \text{ iff } i \in \eta(e).$$

The notion $M,\downarrow e \models \alpha$ can be defined inductively as follows.

(i) $M,\downarrow e \models \alpha$, iff $\alpha \in V(\downarrow e)$, for $\alpha \in \mathscr{P} \cup \mathscr{T}$.
(ii) $M,\downarrow e \models \neg \alpha$, iff $M,\downarrow e \not\models \alpha$.
(iii) $M,\downarrow e \models \alpha \vee \beta$, iff $M,\downarrow e \models \alpha$ or $M,\downarrow e \models \beta$.
(iv) $M,\downarrow e \models \diamondminus_i \alpha$, iff $\exists e' \in E_i : e' \leqslant e$ and $M,\downarrow e' \models \alpha$.

(v) $M,\downarrow e \models \Diamond_i \alpha$ iff $\begin{cases} (e \in E_i): \exists e' \in E_i : e \leqslant e' \text{ and } M,\downarrow e' \models \alpha. \\ (e \notin E_i): \forall e' \in E_i : \text{if } e' \leqslant e \text{ then } M,\downarrow e' \models \Diamond_i \alpha. \end{cases}$

Note that $\diamondminus_i \alpha$ behaves like a normal past modality – it covers all events that lie in the i-past of e. However $\Diamond_i \alpha$ is different: in agent $j, j \neq i$, it asserts that upto the last communication from i, there is a future for agent i satisfying α. In case there is no communication from agent i at all, agent j can assert $\Diamond_i \alpha$ for any formula α.

Define $\boxminus_i \alpha \overset{\text{def}}{=} \neg \diamondminus_i \neg \alpha$ and $\Box_i \alpha \overset{\text{def}}{=} \neg \Diamond_i \neg \alpha$. It can be verified that $\Box_i \alpha \supset \diamondminus_i \Box_i \alpha$ is a valid formula over CSA. It asserts that an invariant formula about an agent must be supported by a communication from that agent. Thus $\Box_i$ is a "strong" modality whereas $\Diamond_i$ is "weak" unlike in standard modal logic. This asymmetry arises from the fact that in distributed systems, the past of other agents can be completely obtained by messages, while the possibilities for the future are only locally known.

Notice that the formula $\tau_i \wedge \tau_j$ is satisfied at a local state $\downarrow e$ only if $\{i,j\} \subseteq \eta(e)$ and thus specifies a synchronization between agents i and j. The infinite set of formulas $\{\tau_i \supset \neg \tau_j \mid i \neq j\}$ together specify that each event is in at most one agent and hence can specify *asynchronous* CSA.

Consider the formula $\diamondminus_i \alpha \wedge \diamondminus_i \beta \supset \diamondminus_i (\alpha \wedge \diamondminus_i \beta) \vee \diamondminus_i (\beta \wedge \diamondminus_i \alpha)$. This specifies that agent i is backwards linear – during a computation if we look back at any two events involving agent i, then they must be ordered. This captures the fact that agents in a CSA are sequential.

Similarly, the formula $\diamondminus_i \alpha \supset \diamondminus_i (\alpha \wedge \boxminus_i (\neg \alpha \supset \boxminus_i \neg \alpha))$ can be used to specify finitary CSA, i.e. those where each event has a finite past. This formula asserts that if α is true somewhere in the past, then we can find an "earliest" point where α is true.

The principal advantage of this logic is that communication between agents in a distributed system can be easily expressed: $\neg \tau_i \wedge \diamondminus_i \alpha \wedge \tau_j$ can be used to specify that i has communicated the truth of α to j sometime in the past.

We shall present a detailed example of reasoning with this logic at the end of this section. First, we present our main technical results for this logic.

We begin with logical system C defined below.

The system C

AXIOM SCHEMES

(A0) All the substitutional instances of the tautologies of Propositional Calculus.
(A1) (a) $\boxminus_i (\alpha \supset \beta) \supset (\boxminus_i \alpha \supset \boxminus_i \beta)$ (Deductive closure)
(b) $\Box_i (\alpha \supset \beta) \supset (\Box_i \alpha \supset \Box_i \beta)$
(A2) (a) $\tau_i \supset (\boxminus_i \alpha \supset \alpha)$ (Local reflexivity)
(b) $\tau_i \supset (\Box_i \alpha \supset \alpha)$
(A3) $\diamondminus_i \diamondminus_j \alpha \supset \diamondminus_j \alpha$ (Transitivity)
(A4) (a) $\diamondminus_i \alpha \supset \Box_i \diamondminus_i \alpha$ (Relating past and future)
(b) $\Diamond_i \alpha \supset \boxminus_i \Diamond_i \alpha$

(A5) $\diamondminus_i\alpha \wedge \diamondminus_i\beta \supset \diamondminus_i(\alpha \wedge \diamondminus_i\beta) \vee \diamondminus_i(\beta \wedge \diamondminus_i\alpha)$. (Backward linearity)
(A6) $\Box_i\alpha \supset \diamondminus_i\Box_i\alpha$ (Communication)
(A7) (a) $\boxminus_i\tau_i$ (Type axioms)
(b) $\tau_i \supset \Box_i\tau_i$

INFERENCE RULES

$$(\mathrm{MP})\frac{\alpha, \alpha \supset \beta}{\beta}, \qquad (\mathrm{TG}\boxminus_i)\frac{\alpha}{\boxminus_i\alpha}, \qquad (\mathrm{TG}\Box_i)\frac{\alpha}{\tau_i \supset \Box_i\alpha}.$$

Axioms A0 to A4 are standard axioms suitably modified to reflect the special interpretation of $\Diamond_i$. A5 asserts that individual agents are sequential. A6 captures the fact that knowledge about another agent's future can only be obtained via communication. A7 ensures that the type propositions from $\mathscr{T}$ are assigned consistently. The rules MP and TG$\boxminus_i$ are standard. The standard form of the rule TG$\Box_i$ will not preserve validity because of the communication requirement imposed by the semantics of $\Box_i$.

Let $\vdash_C \alpha$ denote that α is a thesis of the system C. Let $\models_{CSA} \alpha$ denote that α is valid over the class of models based on CSA. We then have the following result.

Theorem 3.1. *C is a sound and complete axiomatization of the valid formulas of $\mathscr{L}_{CSA}$. In other words $\vdash_C \alpha$ iff $\models_{CSA} \alpha$ for every $\alpha \in \mathscr{L}_{CSA}$.*

When we introduced CSA in §2, we had defined their various subclasses. Let $CSA = (E, \leqslant, \eta)$ be a CSA. Recall that CSA is an n-CSA if $\eta(E) \subseteq \{1, 2, \ldots, n\}$ – that is, there are at most n agents in the system. CSA is an asynchronous-CSA (ACSA) if $\forall e \in E: |\eta(e)| = 1$. CSA is finitary if $\forall e \in E: \downarrow e$ is a finite set. We can combine these notions; for example, an n-ACSA is an ACSA with a bounded number of agents. Similarly, we can have finitary n-CSA, finitary ACSA and, finally, finitary n-ACSA. Figure 10 pictorially represents the relationships between these various classes. The arrows in the figure indicate inclusion.

Let $\mathscr{C}$ denote one of the subclasses of CSA mentioned above. Then we can define the notions of satisfiability and validity relative to $\mathscr{C}$. Thus, a formula α is $\mathscr{C}$-satisfiable if we can find a model $M = ((CSA, LC_{CSA}), V)$ for α such that $CSA \in \mathscr{C}$. We let $SAT_{\mathscr{C}}$ denote the set of $\mathscr{C}$-satisfiable formulas in $\mathscr{L}_{CSA}$. α is $\mathscr{C}$-valid if it is valid over the class of models based on frames in $\mathscr{C}$.

We can axiomatize the $\mathscr{C}$-valid formulas for all these subclasses. The required axiomatizations are obtained by suitably combining the system C with the following axiom schemes.

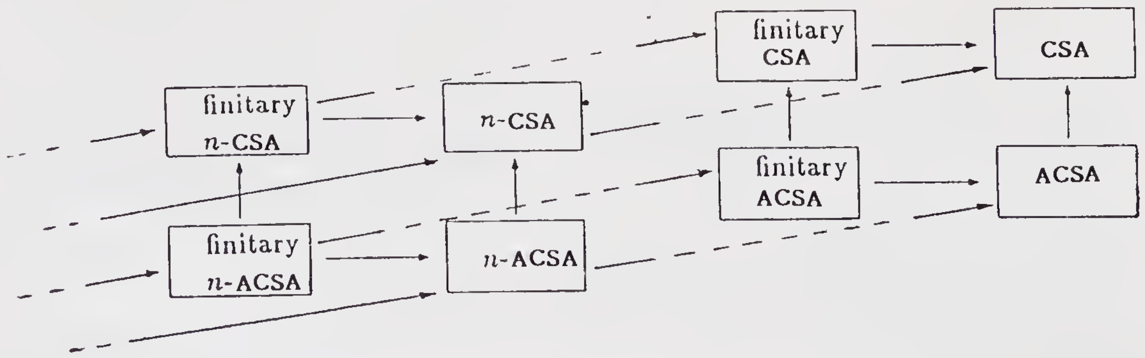


Figure 10. Subclasses of communicating sequential agents.

AUXILIARY AXIOM SCHEMES AND INFERENCE RULES

(A8)	$\tau_1 \vee \tau_2 \vee \ldots \vee \tau_n$	(n agents)
(A9)	$\tau_i \supset \neg \tau_j$, for $i \neq j$	(disjoint agents)
(A10) (a)	$\diamondminus_i \alpha \supset \diamondminus_i(\alpha \wedge \boxminus_i(\neg \alpha \supset \boxminus_i \neg \alpha))$	(well-founded agents and communications)
(b)	$\diamondminus_i \alpha \supset \diamondminus_i(\alpha \wedge \boxminus_j \boxminus_i \neg \alpha)$, for $i \neq j$.	

Theorem 3.2. (1) *The logical system* $C_A \overset{\text{def}}{=} C + \text{(A9)}$ *is sound and complete for the class of models based on* ACSA.
(2) *The logical system* $C_F \overset{\text{def}}{=} C + \text{(A10)}$ *is sound and complete for the class of models based on finitary* CSA.
(3) *The logical system* $C_{FA} \overset{\text{def}}{=} C_A + \text{(A10)}$ *is sound and complete for the class of models based on finitary* ACSA.
(4) *The logical system* $C_n \overset{\text{def}}{=} C + (48)$, $n \in \mathrm{N}$, *is sound and complete fior the class of models based on n*-CSA.
(5) *The logical system* $C_{nA} \overset{\text{def}}{=} C_A + \text{(A8)}$, $n \in \mathrm{N}$, *is sound and complete for the class of models based on n*-ACSA.
(6) *The logical system* $C_{nF} \overset{\text{def}}{=} C_F + \text{(A8)}$, $n \in \mathrm{N}$, *is sound and complete for the class of models based on finitary n*-CSA.
(7) *The logical system* $C_{nFA} \overset{\text{def}}{=} C_{FA} + \text{(A8)}$, $n \in \mathrm{N}$, *is sound and complete for the class of models based on finitary n*-ACSA.

We also have the following relationship between satisfiability in subclasses with an unbounded number of agents and the corresponding subclasses with only a bounded number of agents.

Theorem 3.3. *Let* $\mathscr{C}$ *range over* CSA, ACSA, *finitary* CSA *and finitary* ACSA. *Let* $n\mathscr{C}$, $n \in N$, *denote the corresponding class with a bounded number of agents n. Then* $SAT_{\mathscr{C}} = \cup_n SAT_{n\mathscr{C}}$.

We now give a detailed example of how communication between agents can be specified in $\mathscr{L}_{CSA}$. Consider a distributed database accessed by n processes which communicate with each other by exchanging messages. A protocol is needed whereby the processes can *commit* to a distributed transaction. When each committed process knows that all the others have also committed it can go ahead and perform its local share of the distributed transaction. For this, the following requirement must be met.

If any process commits to the transaction then it eventually knows that all processes in the system have also committed.

Such *distributed transaction commit protocols* commonly arise in the design of distributed systems (Pinter & Wolper 1984, pp. 28–37).

We now specify the protocol requirement in our logical language. Let $\{c_1, \ldots, c_n\}$ be a set of atomic propositions, where c_j is read to mean "process j has committed to the transaction". The formula

$$\bigwedge_i \left(\tau_i \wedge c_i \supset \Diamond_i\left(\bigwedge_j \diamondminus_i c_j\right)\right), \tag{1}$$

expresses the requirement above.

A two-stage implementation of this protocol may use two local boolean variables in each process P_i:

- a variable l_i in which process P_i records whether it can participate in the transaction or not, and
- a variable, which we also call c_i, to record the commitment of the process to the transaction.

The implementation can perhaps run as follows:
Process P_i:

(1) as soon as a local decision l_i is made, broadcast l_i to all other processes;
(2) when l_j is heard from all j, set c_i to *True*;
(3) as soon as c_i is set, broadcast it to all other processes;
(4) when c_j is heard from all j, perform transaction;
(5) acknowledge all incoming messages.

All processes follow the same protocol in a symmetric manner. This is, of course, a naïve protocol. However, our aim here is to merely illustrate the use of our logical language. Let us again, by abuse of notation, use $\{l_1, \ldots, l_n\}$ to denote another set of atomic propositions. Consider now the following formulas:

$$\bigwedge_i \left(\tau_i \supset \left(c_i \equiv \bigwedge_j \diamondminus_j l_j \right) \right), \tag{2}$$

"c_i is set *True* only when l_j is heard from all other processes P_j",

$$\bigwedge_i \left(\tau_i \wedge c_i \supset \Diamond_i \bigwedge_j \diamondminus_j \diamondminus_i c_i \right), \tag{3}$$

"if c_i is set, then it will be broadcast and acknowledged".

Note that here an agent has to assert something about the state of other agents and this can be done only using messages from them. The formula $\Diamond_i \diamondminus_j \diamondminus_i c_i$ says that agent i has received an acknowledgement from agent j of the message c_i sent from i to j. This is necessary because we assume that messages may be lost in this network.

It is easy to verify that the formulas (2) and (3) together imply the requirement (1) above. In fact, we can use the axiom system C and logically deduce the requirement from (2) and (3). This verifies that the simple protocol above meets its specification.

Note that the protocol above works for only one transaction, in the sense that the commitment is *stable*; once a process commits to the transaction, it stays committed. When a protocol is needed for several transactions, we can index the transactions by sequence numbers and modify the specification above appropriately.

While the preceding example illustrates the specification of a protocol which assumes complete connectivity in the network of communicating agents, we can also specify protocols which demand specific patterns of connectivity. Since agents are syntactically mentioned in formulas, this logic is particularly suited for describing communications which name specific agents. We illustrate this point with another detailed example.

Assume that processes $P_0, P_1, \ldots, P_{n-1}$ are connected in a ring and communicate with each other only by exchanging messages. A process P_i can communicate only with its neighbours P_{i-1} and P_{i+1} on the ring. Here and in the sequel, addition and subtraction are assumed to be *modulo n.*

Assume that each process P_i maintains a variable x_i taking values in N and whose value initially is v_i, for $0 \leqslant i \leqslant n-1$. It is described to specify a distributed protocol which computes the greatest common divisor (GCD) of the values $v_0, \ldots, v_{n-1}$. Let *result* denote the value of the constant $gcd(v_0, v_1, \ldots, v_{n-1})$. When the computation terminates, the variables x_i, $i \in \{0, \ldots, n-1\}$ should satisfy

$$x_0 = x_1 = \ldots = x_{n-1} = result.$$

Since our logical language is propositional in nature we cannot express values of variables and hence assume countably many propositions X_i^k, $k \in$N, to denote "$x_i = k$". With this understanding we write such propositions as equalities. Similarly we assume propositions to denote "$k < l$", "$k = i - j$" etc. The protocol requirement is then specified by

$$\bigwedge_i (\tau_i \wedge (x_i = v_i) \supset \Diamond_i \bigwedge_k \diamondminus_k (x_k = result)).$$

An algorithm for computing the GCD can be described as follows: process P_i, at any state, compares the current value of x_i, with the current values of its neighbours, x_{i-1} and x_{i+1}. In case x_i is smaller, nothing needs to be done; if x_{i-1} is smaller, x_i is updated to be $x_i - x_{i-1}$; similarly, if x_{i+1} is smaller, x_i is updated to be $x_i - x_{i+1}$. Whenever the value of x_i changes, this is communicated to the neighbouring processes. Eventually, all values stabilize at the greatest common divisor.

As before, we assume that messages may fail and hence received messages are always acknowledged. Let $\Diamond_{i \to j} \alpha$ abbreviate the formula $\tau_i \wedge \Diamond_i \diamondminus_j \diamondminus_i \alpha$. (In some sense, this stands for "*i* sends the message α to *j* and receives an acknowledgement".)

Our protocol can now be specified as

$$\bigwedge_i (\tau_i \supset \Box_i \delta \wedge \boxminus_i \delta)$$

where $\delta \overset{\text{def}}{=} \delta_0 \wedge \delta_1 \wedge \delta_2 \wedge \delta_3$ is given by:

$$\delta_0 : (x_i = v \supset \bigwedge_{j \in \{i-1, i+1\}} \Diamond_{i \to j} (x_i = v))$$

"neighbours are always kept informed of current x_i value"

$$\delta_1 : (x_i = v \supset \Box_i (x_i = v' \supset v' \leqslant v))$$

"values are never increased"

$$\delta_2 : (x_i = v \wedge \diamondminus_{i-1} (x_{i-1} = v') \wedge v' < v \supset \Diamond_i (x_i = v'' \wedge v'' = v - v'))$$

"if $x_{i-1} < x_i$ then $x_i := x_i - x_{i-1}$"

$$\delta_3 : (x_i = v \wedge \diamondminus_{i+1} (x_{i+1} = v') \wedge v' < v \supset \Diamond_i (x_i = v'' \wedge v'' = v - v'))$$

"if $x_{i+1} < x_i$ then $x_i := x_i - x_{i+1}$".

It is easy to see that this specifies a distributed implementation of Euclid's algorithm for computing the GCD.

4. Discussion

In this paper, we have looked at models for distributed systems which emphasize their nonsequential behaviour and considered their logical characterization using an assortment of modal logics.

A fair amount of theory has been developed for the models we have considered. Our notion of a distributed transition system is only one of several that have been considered; alternative formulations include those of Degano & Montanari (1987) and Boudol & Castellani (1988). Stark (1989) had defined a related class of models called concurrent transition systems. In net theory, more general net systems include Petri nets, predicate/transition nets and coloured nets (Brauer *et al* 1987). As far as event structures are concerned, we have only considered prime event structures in this paper; other classes of event structures include stable event structures and general event structures (Winskel 1987. pp. 325–392) as well as flow event structures (Boudol 1990, pp. 62–95). Systems of communicating sequential agents were introduced in Lodaya *et al* (1989b), as a generalization of the n-agent event structures described in Lodaya & Thiagarajan (1987, pp. 290–303).

The models that we have dealt with in this paper are closely related to each other. We have described how labelled net systems and labelled event structures give rise to distributed transition systems in a natural way. A strong relationship also exists between elementary net systems and prime event structures (Nielsen *et al* 1980, 1990). The connection between CSA and event structures is described in Lodaya *et al* (1989b). By establishing formal connections between models in this manner, we can translate results obtained using one class of models to other classes.

As for the logics that we have described here, the main results that we have are sound and complete axiomatizations for different classes of models (see Lodaya *et al* 1987, 1989a, pp. 508–522, 1989b, 1991, Mukund & Thiagarajan 1989, pp. 143–160, 1991, and Mukund 1990). For the logic for distributed transition systems, we also have various decidability and undecidability results (Lodaya *et al* 1991). However, for the logics for event structures and CSA, the decidability question remains open.

Several attempts have been made to use logics to characterize the behaviour of distributed programs. Temporal modalities have been traditionally interpreted over different types of tense structures (Burgess 1980, 1984, pp. 89–133). Using the interleaving approach to modelling concurrency, various authors have used temporal logics defined on sequences and trees to describe concurrent computations (see e.g. Pnueli 1977, pp. 46–57, Gabbay *et al* 1980; Clarke *et al* 1986). Pinter & Wolper (1984, pp. 28–37) have extended this work to true concurrency by explicitly using partial orders to represent concurrent computations. Katz & Peled (1989, pp. 489–507) have defined a first-order temporal logic over sets of partial orders.

However, the use of classes of behavioural structures for distributed systems as frames for logics seems to be relatively new. Penczek (1988) has used event structures as frames and is the first to use an explicit modality to represent conflict. Reisig (1986, pp. 603–627) is working on logics which directly use elementary net systems as frames. Christiansen (1989) has worked with CSA-like frames; he uses an indexed $\triangle$ modality in his logic to describe concurrency across agents.

Trace theory is a language theoretic approach to describing concurrency which we have not considered. This formalism also gives rise to models of distributed systems with true concurrency. Here, along with an alphabet of actions, one is given an *independence relation* declaring which actions in the system are concurrent. Instead of viewing a computation as a string of symbols from the alphabet, one now considers sequences made up of sets of concurrent actions (sequences of concurrent steps, in our framework), which are called *traces*. Like strings, traces form a monoid, called a partially commutative monoid, and so one can meaningfully talk about trace languages. A syntactic Kleene-like characterization of regular trace languages has been given by Ochmanski (1985), while a characterization in terms of automata has been obtained by Zielonka (1987). The *pomsets* of Gischer and Pratt (Pratt 1986) are similar to traces.

Logics for trace theory have not been considered in the literature. We believe that results like the ones in § 3.1 can be obtained (Lodaya *et al* 1991).

Another widely prevalent approach to modelling concurrency is *algebraic*. One way of describing sequential nondeterministic programs is through regular expressions, by interpreting the operators $\bullet$, $+$ and $*$ as sequential composition, choice and iteration. Similarly, in the algebraic approach to concurrency, one introduces an operator to denote the parallel composition of programs. Program behaviour is specified by modelling the language operators in an appropriate semantic domain. Popular languages for concurrency include CSP (Hoare 1984), CCS (Milner 1989) and ACP (Bergstra & Klop 1984), and the models most often used are transition systems (Plotkin 1981) and equational algebras (Bergstra & Klop 1984). Most of this work has been based on interleaving models and only recently have attempts been made to give a "truly concurrent" semantics to these languages (Olderog 1987, pp. 196–223, van Glabbeek & Vaandrager 1987, pp. 224–242, Degano *et al* 1989, pp. 438–466). An earlier denotational semantics using event structures as domains was given in Winskel (1982, pp. 561–577).

In this framework, Hennessy & Milner (1985) have used action-indexed logics to characterize computations of sequential nondeterministic systems. Assuming an interleaving model of concurrency, this characterization extends to the computations of distributed systems. This work has been considerably extended by Stirling (1987). However, the emphasis here is on axiomatizing *program equivalences* using equational logic. Our use of action-indexed logics for models exhibiting true concurrency is inspired by this work, but we have concentrated on axiomatizing the valid formulas, as is traditional in logic.

Logics in which the modalities are indexed by programs, rather than just actions, arose in the framework of program verification (Hoare 1969). Programs with parallel composition operators have been considered by several authors (e.g. Apt *et al* 1980, Moitra 1983). Dynamic logics, originally defined over sequential programs (Harel 1984), have been extended with an operator for intersection to model synchronization (Peleg 1987). However, a lot of work remains to be done on characterizing models for true concurrency using program-indexed logics.

References

Apt K R, Francez N, de Roever W P 1980 A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.* 2: 359–385

Bergstra J A, Klop J W 1984 Process algebra for synchronous communication. *Inf. Control* 60(1–3): 109–137

Boudol G 1990 *Flow event structures and flow nets. Lecture Notes in Computer Science. Vol. 469* (Berlin: Springer-Verlag) pp. 62–95

Boudol G, Castellani I 1988 A non-interleaving semantics for CCS based on proved transitions. *Fundam. Inf.* 11: 433–452

Brauer W, Reisig W, Rozenberg G (eds) 1987 *Petri nets: central models and their properties. Lecture Notes in Computer Science. Vol. 254* (Berlin: Springer-Verlag)

Burgess J P 1980 Decidability for branching time. *Stud. Logica* 39: 203–218

Burgess J P 1984 Basic tense logic. In *Handbook of philosophical logic II* (eds) D Gabbay, F Guenthner (Dordrecht: D Reidel) pp. 89–133

Christiansen S 1989 *A logical characterization of linear n-agent event structures*, M Sc thesis, Computer Science Department, Århus Univ. Århus

Clarke E M, Emerson E A, Sistla A P 1986 Automatic verification of finite-state concurrent programs using temporal logic specifications. *ACM Trans. Program. Lang. Syst.* 8: 244–263

Degano P, Montanari U 1987 Concurrent histories: A basis for observing distributed systems. *J. Comput. Syst. Sci.* 34: 422–461

Degano P, de Nicola R, Montanari U 1989 *Partial ordering descriptions and observations of nondeterministic concurrent processes. Lecture Notes in Computer Science. Vol. 354* (Berlin: Springer-Verlag) pp. 438–466

Gabbay D, Pnueli A, Shelah S, Stavi J 1980 On the temporal analysis of fairness. *Proc. 7th ACM Symp. Principles of Program. Lang.* (New York: ACM Press) pp. 163–173

van Glabbeek R, Vaandrager F 1987 *Petri net models for algebraic theories of concurrency. Lecture Notes in Computer Science. Vol. 259* (Berlin: Springer-Verlag) pp. 224–242

Harel D 1984 Dynamic logic. In *Handbook of philosophical logic II* (eds) D Gabbay, F Guenthner (Dordrecht: D Reidel) pp. 497–604

Harel D 1985 Recurring dominoes: making the highly undecidable highly understandable. *Ann. Discrete Math.* 24: 51–72

Hennessy M, Milner R 1985 Algebraic laws for nondeterminism and concurrency *J. Assoc. Comput. Mach.* 32: 137–161

Hoare C A R 1969 An axiomatic basis for computer programming. *Commun. ACM* 12: 576–580, 583

Hoare C A R 1984 *Communicating sequential process* (New York: Prentice-Hall)

Katz S, Peled D 1989 *An efficient verification method for parallel and distributed programs. Lecture Notes in Computer Science. Vol. 354* (Berlin: Springer-Verlag) pp. 489–507

Lodaya K, Parikh R, Ramanujam R, Thiagarajan P S 1992 A logical study of distributed transition systems, Report IMSc/92/07, Inst. Math. Sci., Madras

Lodaya K, Ramanujam R, Thiagarajan P S 1989a *A logic for distributed transition systems. Lecture Notes in Computer Science. Vol. 354* (Berlin: Springer-Verlag) pp. 508–522

Lodaya K, Ramanujam R, Thiagarajan P S 1989b Temporal logics for communicating sequential agents: *Int. J. Found. Comput. Sci.* (to appear)

Lodaya K, Thiagarajan P S 1987 *A modal logic for a subclass of event structures. Lecture Notes in Computer Science. Vol. 267* (Berlin: Springer-Verlag) pp. 290–303

Mazurkiewicz A 1989 *Basic notions of trace theory. Lecture Notes in Computer Science. Vol. 354* (Berlin: Springer-Verlag) pp. 285–363

Milner R 1989 *Communication and concurrency* (New York: Prentice-Hall)

Moitra A 1983 (Letter) *ACM Trans. Program. Lang. Syst.* 5: 500–501

Mukund M 1990 Expressiveness and completeness of a logic for well branching prime event structures, Report TCS-90-1, School of Math., SPIC Science Foundation, Madras

Mukund M, Thiagarajan P S 1989 *An axiomatization of event structures. Lecture Notes in Computer Science. Vol. 405* (Berlin: Springer-Verlag) pp. 143–160

Mukund M, Thiagarajan P S 1991 A logical characterization of well branching event structures. *Theor. Comput. Sci.* (to appear)

Nielsen M, Plotkin G, Winskel G 1980 Petri nets, event structures and domains I. *Theor. Comput. Sci.* 13: 86–108

Nielsen M, Rozenberg G, Thiagarajan P S 1990 Behavioural notions for elementary net systems. *Distrib. Comput.* 4: 45–57

Ochmanski E 1985 *Regular trace languages*, Ph D thesis, University of Warsaw, Warsaw

Olderog E-R 1987 *Operational Petri net semantics for CCSP. Lecture Notes in Computer Science. Vol. 266* (Berlin: Springer-Verlag) pp. 196–223

Parikh R 1989 Decidability and undecidability in distributed transition systems. In *A perspective in theoretical computer science – commemorative volume for Gift Siromoney* (ed) R Narasimhan (Singapore: World Scientific) pp. 199–209

Peleg D 1987 Concurrent dynamic logic. *J. Assoc. Comput. Mach.* 34: 450–479

Penczek W 1988 A temporal logic for event structures. *Fundam. Inf.* 11: 297–326

Pinter S, Wolper P 1984 A temporal logic for reasoning about partially ordered computations. *Proc. 3rd ACM Symp. Principles of Distrib. Comput.* (New York: ACM Press)

Plotkin G 1981 A structural approach to operational semantics, Report DAIM IFN-19, Computer Science Dept, Århus Univ, Århus

Pnueli A 1977 The temporal logic of programs. *Proc. 18th IEEE Conf. Foundations of Comput. Sci.* (New York: IEEE) pp. 46–57

Pratt V 1986 Modelling concurrency with partial orders. *Int. J. Parallel Programming* 15: 33–71

Reisig W 1986 *Towards a temporal logic for causality and choice in distributed systems. Lecture Notes in Computer Science. Vol. 354* (Berlin: Springer-Verlag) pp. 603–627

Stark E W 1989 Concurrent transition systems. *Theor. Comput. Sci.* 64: 221–269

Stirling C 1987 Modal logics for communicating systems. *Theor. Comput. Sci.* 49: 311–347

Thiagarajan P S 1990 Some behavioural aspects of net theory. *Theor. Comput. Sci.* 71: 133–153

Wang H 1961 Proving theorems by pattern recognition II. *Bell Syst. Tech. J.* 40: 1–41

Winskel G 1982 *Event structure semantics for CCS and related languages. Lecture Notes in Computer Science. Vol. 140* (Berlin: Springer-Verlag) pp. 561–577

Winskel G 1987 *Event structures. Lecture Notes in Computer Science. Vol. 255* (Berlin: Springer-Verlag) pp. 325–392

Zielonka W 1987 Notes on finite asynchronous automata. *RAIRO Inf. Theor. Appl.* 21(2): 99–135

Sādhanā, Vol. 17, Part 1, March 1992, pp. 167–191. © Printed in India.

Levels of knowledge in distributed systems

ROHIT PARIKH[1] and PAUL KRASUCKI[2]

[1]Department of Computer Science, CUNY Graduate Center, 33 West 42nd Street, New York, NY 10036, USA
[2]Department of Computer Science, Rutgers University, Camden College of Arts and Sciences, Camden, NJ 08102, USA

Abstract. We correlate the *level* of knowledge of certain formulas in a *group of individuals* with certain regular, downward closed, sets of strings. We show that in suitable circumstances, all such sets can occur as levels of knowledge but that the lack of synchrony, or the lack of asynchrony when there are only two processors in the group, can create more or less severe restrictions.

Keywords. Levels of knowledge; distributed systems; common knowledge.

1. Introduction

It has been suggested recently that the notions of knowledge and common knowledge may be useful in analysing the behaviour of distributed systems and in designing protocols (Parikh & Ramanujam 1985, pp. 256–268; Chandy & Misra 1986; Parikh 1986, pp. 322–331; Halpern & Zuck 1987, pp. 269–280; Moses & Tuttle 1988; Halpern & Moses 1990).

As specific examples, we cite the paper by Moses & Tuttle (1988) who proved that certain synchronized action problems require common knowledge, and that there is always a most efficient solution which is an implementation of a simple knowledge based algorithm – an algorithm where there are explicit tests of knowledge. This algorithm is of the form "repeat... until $C_U A$" for a certain formula A.[1]

Again, it was shown by Halpern & Zuck (1987, pp. 269–280) that if a sequence of bits is communicated in an asynchronous system where messages can be delayed or lost (but if they are received, they are received in order), then to prove correctness of the protocol it is necessary and sufficient to prove that $K_s K_r K_s K_r$("*value of the current bit*")[2] is true whenever the sender sends the next bit. Unlike the case with Moses & Tuttle (1988), common knowledge is not necessary, nor would it be attainable if needed.

[1] $C_U A$ means that there is common knowledge among the processes in U that A is true. The following important property of common knowledge is used in the synchronization algorithms: common knowledge is always achieved *simultaneously* by all the processes involved.

[2] The sender knows that the receiver knows that the sender knows that the receiver knows the value of the current bit.

Other places where these notions have been found to be important are the semantics of natural language (Lewis 1969; Schiffer 1972) and mathematical economics (Aumann 1976; Parikh & Krasucki 1990).

Since there are useful protocols which can be specified in terms of knowledge formulae, formulae of the form $K_{i_1}K_{i_2}...K_{i_m}A$, where K_{i_j} are knowledge operators, we investigate which sets of such formulae can occur as states (or *levels*) of knowledge for some formula A.

We define a logic of knowledge by augmenting some base logic L (e.g. propositional logic) with modal operators K_i for $i \leqslant n$. For every process i, i knows A if and only if according to i's view of the world, A must be true.

We assume that processes' information is always correct, although often incomplete. Processes may not be able to describe precisely the global state of the system, some states may be indistinguishable to them, but they always admit the real state of the system as one of the possibilities.

This leads to a Kripke semantics where every process i has some indistinguishability relation R_i and in the state s, process i knows that a formula A is true iff for all the states s' which look the same to i as s, A is true in s'. Since we deal with distributed systems, and we want to carry all the information about the past in our states, we will call the states *histories*. A history H is a sequence of snapshots of the system, i.e. a sequence of n-tuples of actions (n is the number of processes), where a snapshot is taken at every tick of a very fine clock. If a process does nothing, its action is taken to be the special *null* action.

A first natural problem is the characterization of the sets of strings of knowledge operators x such that for some fixed formula A and history H, $H \models xA$.

This is a purely logical problem. The results are valid for every model of distributed systems where the indistinguishability relation R_i for every process i is an equivalence relation.

These sets of knowledge formulae (formulae of the form $K_iK_j....A$), we will call *levels* of knowledge.

The second problem is the design of a protocol, when possible, in which all the knowledge formulae (and *exactly* the knowledge formulae) in the set are satisfied.

To solve this problem we need to design a model of distributed system in which we can control the acquisition of knowledge. We need a system in which there is no accidental knowledge (or accidental synchrony), a system in which all the knowledge of the process about the others must be a result of some communications.

If processes in a distributed system know each other's programs, the lack of knowledge in the system may be due to: non-determinism, inputs, faults, communications and asynchrony.

In our system, lack of knowledge will be the result of three factors: initial private inputs of processes, lack of synchrony, possible delays in the communication system.

It turns out that the existence of a protocol depends on different assumed means of communication available in a distributed system.

We will analyse systems where all communications are asynchronous (there is uncertainty about the delivery time of a message); systems where all communications are instantaneous (synchronous systems); and systems where both types of communications are available.

The paper will consist of three main parts:

(1) *Logic and levels of knowledge* – description of the logic, definition of the level of knowledge of a formula, characterization of levels of knowledge.

(2) *Model of a distributed system* – definition of the distributed system in which all knowledge about other processors is acquired through communication.
(3) *Realization of levels of knowledge in distributed systems* – given a set which may be the level of knowledge of some formula we design a protocol which realizes precisely that level.

Our work is a continuation and extension of the research initiated by Parikh (1986, pp. 322–331) and it includes the results from Parikh (1986, pp. 322–331) as well as Parikh & Krasucki (1986).

2. Logic and levels of knowledge

A protocol is a set of global histories where global histories are finite sequences of events in the system, which are prefixes of runs of the system. L_0 is a language which describes properties of the global histories in protocol **P**. For every sentence A in L_0, and for every history $H \in \mathbf{P}$, A is either true or false in H.

We want to make sure that every processor's private information is expressible in our language. To accomplish that we assume that we have in our language a countable set of propositions $Q_{i,j}$, where $Q_{i,j}$ is the proposition that the jth input value of processor i is 1. The $Q_{i,j}$ are independent but only finitely many can be true. The truth values of the $Q_{i,j}$ are the private facts of individual i. When we do not care about the 'owners' of these facts $Q_{i,j}$ then we shall refer to them just as the 'private facts' and denote them as P_k.

L is the closure of L_0 under truth functional connectives. L can be extended to a larger language L_K which is the closure of L under the knowledge operators K_i (for $i \in N$) and the usual truth functional connectives. Here $K_i(A)$ means that processor i knows A.

The class of all models we consider is the class of all protocols **P** as described in the next section. Let's fix **P**. Now we define the notion $H \models A$ for A in L_K by recursion on the complexity of A. We assume that for every process there is some equivalence relation $\approx_i$ defined on histories. $H \approx_i H'$ means that according to i's *view* H and H' are indistinguishable.

(0) If A is from L_0 then the semantics is given.
(1) If A is $Q_{i,j}$ then A is true in H if the jth bit of the input of processor i in H is 1:

$$H \models A, \quad \text{iff } H = (v_1, \ldots, v_n); H', (v_i)_j = 1.$$

(2) A is $\neg A'$ then $H \models A$, iff $H \not\models A'$.
If A is $B \vee C$ then $H \models A$, iff ($H \models B$ or $H \models C$).
(3) $H \models K_i A$, iff $\forall H' \in \mathbf{P}\ H \approx_i H' \rightarrow H' \models A$.

We will sometimes also need to refer to common knowledge operators C_U where $U \subseteq N$. If A is of the form $C_U(B)$, then A is true in H iff B is true in every H' such that there is a chain of processors $i_1, \ldots, i_k$ in U and a chain of histories $H = H_1, \ldots, H_k, H_{k+1} = H'$, such that for every $1 \leqslant j \leqslant k, H_j \approx_{i_j} H_{j+1}$.

We also define a new relation $\approx_U$ as follows:

$$H \approx_U H' \text{ iff } \exists i \in U\ H \approx_i H'.$$

Then $\approx_U^*$ will be the reflexive, transitive closure of $\approx_U$, and we have the following equivalent semantics for the common knowledge operator:

$$H \models C_U A, \quad \text{iff } \forall H' \in \mathbf{P}, \quad H \approx_U^* H' \rightarrow H' \models A.$$

Note that if U is empty, then $H \models C_U A$ iff $H \models A$, since $\approx_U^*$ is then the identity relation. For a set X of formulas, we will write $H \models X$ to mean that for all $A \in X$, $H \models A$.

Since the $\approx_i$ are equivalence relations, the following S5 axioms and rules hold.

(1) All tautologies.
(2) $K_i(A) \wedge K_i(A \rightarrow B) \rightarrow K_i(B)$.
(3) $K_i(A) \rightarrow A$.
(4) $K_i(A) \rightarrow K_i(K_i(A))$.
(5) $\neg K_i(A) \rightarrow K_i(\neg K_i(A))$.

The rules *modus ponens* and necessitation are sound where the latter allows us to infer a formula $K_i(A)$ provided that we have shown that the formula A holds for *all* histories H. This logic in the case when L_0 is propositional is known as $LK5$.

The same axiom system will also be sound if we replace K_i by C_U. In this case we can add one more axiom:

(6) $V \subseteq U \rightarrow (C_U A \rightarrow C_V A)$

If $U = \{i\}$ then $\approx_i$ and $\approx_U^*$ coincide since $\approx_i$ is transitive.

In the following characterization of levels of knowledge we will not need all the axioms: axiom 5 will not be used, so our results are valid also for the system in which $\approx_i$ are reflexive and transitive but not necessarily symmetric (the logic satisfies S4 axioms).

2.1 *Levels of knowledge*

In this section we show how to define the *level of knowledge* of a formula as a set of strings over the alphabet Σ where $\Sigma = \{K_1, \ldots, K_n\}$ and investigate properties of levels of knowledge.

Consider a formula A and some global history H. The formula may be true but not known to anyone. In that case we have $H \models A$ but not $H \models K_i(A)$ for any i. Or perhaps it may be known to some i that A is true. In the latter case, $K_i(A)$ is true and A will hold at all histories H' such that $H' \approx_i H$. The formula $K_j K_i(A)$ expresses the stronger assertion that not only is A known to i, but also that this fact is known to j. Still more is known in the system if among i and j both know that both know, ..., that both know A. This is common (or mutual) knowledge of A between i and j and is denoted by $C_{\{i,j\}}(A)$. Thus a formula that is known may be known at a higher (in some sense) or lower level.

The highest possible level of knowledge here is $C_N(A)$, A is *common knowledge for the whole group*, which holds if for all strings x of knowledge operators, xA holds. We shall give now the precise definition of the level of knowledge, but first let's look at the set $T(A, H)$ of all strings of knowledge operators x such that xA is true in H:

$$T(A, H) = \{x \mid x \in \Sigma^* \text{ and } H \models xA\}.$$

If there is some non-empty string x_0 in $T(A, H)$, then $T(A, H)$ is infinite. This is a consequence of the following theorem:

Theorem 1. (a) *Let Σ be the alphabet whose symbols are $\{K_1,\ldots,K_n\}$. For all a in Σ, and for all x, y, in Σ^*, and all formulae A,*

$$\vdash xayA \leftrightarrow xaayA,$$

and hence for all H, $H \vDash xayA$ iff $H \vDash xaayA$. That is, repeated occurrences of a are without effect and if $xay \in T(A,H)$ then $\forall n\ xa^n y \in T(A,H)$.
(b) *For a subset U of $\{1,\ldots,n\}$, let $\Delta = \{K_i | i \in U\}$. Then for all histories H, and formulae A, $H \vDash C_U(A)$ iff for all strings $x \in \Delta^*$, $H \vDash xA$.*

Proof. (a) is straightforward using the fact that $\approx_i$ is an equivalence relation. Let $a = K_i$. Then $H \vDash ayA$ iff $\forall H', H' \approx_i H, H' \vDash yA$. But by transitivity of $\approx_i$ this yields: $\forall H', H' \approx_i H, \forall H'', H'' \approx_i H', H'' \vDash yA$, i.e. $H \vDash aayA$. Since ayA and $aayA$ are equivalent, it is easily seen that so are $xayA$ and $xaayA$. The proof of (b) follows from the fact that the relation $\approx_\Delta$ is the transitive closure of $\cup(\approx_i): i \in \Delta$. □

Since occurrences of substrings of the form $K_i K_i$ don't carry any more information than strings K_i we define levels of knowledge of formulae by excluding all strings containing consecutive occurrences of the same knowledge operator.

DEFINITION 1

(a) A string x is *simple* if x contains no substrings $K_i K_i$.
(b) Given a formula A and a history H, the *level* $L(A,H)$ of A at H is the set of all simple x in Σ^* such that $H \vDash xA$.

If H is clear from the context, or not important, then we shall drop it as a parameter. The set of simple strings on an alphabet Σ will be denoted Σ^s. Thus $L(A,H)$ will always be a subset of Σ^s.

2.2 *Embeddability*

Now we will try to characterize levels of knowledge. First we need to introduce the embeddability ordering on strings which turns out to be important.

DEFINITION 2

Given two strings x and y, we say that x is *embeddable* in y ($x \leqslant y$), if all the symbols of x occur in y, in the same order, but not necessarily consecutively. Formally: Let $x = a_1 \cdots a_m$ and $y = b_1 \cdots b_p$. Then x is embeddable in y iff there is a function f from $\{1,\ldots,m\}$ to $\{1,\ldots,p\}$ such that $\forall i < j \leqslant m, f(i) < f(j)$ and $a_i = b_{f(i)}$.

Thus the string *aba* is embeddable in itself, in *aaba* and in *abca*, but not in *aabb*.

Properties of the embeddability relation $\leqslant$

Fact 1. Embeddability is a well partial order, i.e. it is not only well founded, but every linear order that extends it is a well order (equivalently, it is *well founded* and every set of mutually incomparable elements is finite).

Fact 2. Embeddability can be tested in linear time by a two tape Turing machine.

For a proof of fact 1, see Higman (1952) and deJongh & Parikh (1977). Fact 2 is straightforward.

We also need a weaker (larger) relation defined on Σ^*, which we call *K-embeddability*.

DEFINITION 3

We define the *K-embeddability* relation $\preccurlyeq$ as follows:

If $x = a_1 \cdots a_m$ and $y = b_1 \cdots b_p$ are elements of $\{K_1, \ldots, K_n\}^*$, then x is *K-embeddable* in y iff there is a function f from $\{1, \ldots, m\}$ to $\{1, \ldots, p\}$ such that $\forall i \leqslant j \leqslant m, f(i) \leqslant f(j)$ and $a_i = b_{f(i)}$.

The condition defining $\preccurlyeq$ is weaker than that defining $\leqslant$. Hence the relation $\preccurlyeq$ extends the relation $\leqslant$ so that $aaba \preccurlyeq aba$, but $aaba \nleqslant aba$. However, for all simple x and for all y, $x \leqslant y$ iff $x \preccurlyeq y$. Given a string x, there is a shortest simple y such that $x \preccurlyeq y$. We shall denote y as $Sim(x)$, the simplification of x. For example, if $x = abbacbbaa$ then $Sim(x) = abacba$.

DEFINITION 4

A *downward closed* subset of $\Sigma^s = \{K_1, \ldots, K_n\}^s$ is a subset X such that if $x \in X$ and $y \preccurlyeq x$, then $y \in X$.

Theorem 2. *For all strings $x \preccurlyeq y \in \Sigma^s$, all formulae A and for all histories H, if $H \models yA$ then $H \models xA$.*

Proof. We use induction on the sum of lengths of x and y. If the sum is 0, then the lemma is immediate. Otherwise y must be nonempty and let y be $K_i y'$. Now either $x \preccurlyeq y'$ or x is $K_i x'$ for some i and $x' \preccurlyeq y'$. In the first case yA implies $y'A$ which (by induction hypothesis) implies xA. In the second case, $y'A$ implies $x'A$ by induction hypothesis, and therefore by necessitation $K_i y' A$ implies $K_i x' A$, so we get yA implies xA. □

COROLLARY 1

Every level of knowledge is a set of simple strings, downward closed with respect to the order $\preccurlyeq$.

COROLLARY 2

The complement of every level of knowledge is upward closed with respect to $\preccurlyeq$.

So far we have a necessary condition for the set of strings of knowledge operators to be the level of knowledge of some formula in some history. We can infer for example that there is no formula A and history H, such that $H \models K_2 K_1 A$ and $H \models \neg K_2 A$. This is because if $K_2 K_1 \in L(A, H)$ then since $L(A, H)$ is downward closed, K_2 is also in it. In this case, we could also have seen this fact directly by deriving $K_2 A$ from $K_2 K_1 A$, but other cases might be more subtle. Thus we will need the notion of the smallest downward closed set of strings of knowledge operators including the given set X. We will call this set the downward closure of X and denote it as $dc(X)$. We start by investigating some properties of the operation of downward closure.

2.3 *Downward closures*

DEFINITION 5

The *downward closure* $dc(X)$, of $X \subseteq \Sigma^s$ is the smallest set $Y \subseteq \Sigma^s$ such that $X \subseteq Y$ and for all x, if $x \in Y$ and $y \preccurlyeq x$ then $y \in Y$.

Properties of downward closure
Facts 3 and 4 depend only on the fact that $\preccurlyeq$ is a partial order.

Fact 3. If B is downward closed then for all A, $A \subseteq B$ iff for every $x \in A$ there is $y \in B$ such that $x \preccurlyeq y$.

Fact 4. $dc(A \cup B) = dc(A) \cup dc(B)$

Fact 5. $dc(A; B) = Sim(dc(A); dc(B))$ where, for $X \subseteq \Sigma^*$, $Sim(X) = \{Sim(x) | x \in X\}$. Obviously $Sim(X) \subseteq \Sigma^s$.

In future, we will omit the operator *Sim* in contexts where non-repeating strings are the only strings involved, stipulating that for subsets X, Y of Σ^s, X; Y will indicate $Sim(X; Y)$ where the second ';' denotes concatenation of X, Y as subsets of Σ^*. A^s will denote $Sim(A^*)$.

Fact 6. $dc(A^s) = U^s$ where $U = \{\sigma | \sigma \in \Sigma, \exists x \in A \ \sigma \preccurlyeq x\}$.

Facts 3, 4 and 5 are straightforward. To prove fact 6 first notice that for any $U \subseteq \Sigma$, U^s is downward closed, and clearly $dc(A^s) \subseteq U^s$. To show that $U^s \subseteq dc(A^s)$ let's assume that in fact there is some string $x \in \Sigma^s$, such that $x \in U^s$ and $x \notin dc(A^s)$. Say $x = a_1 a_2 \cdots a_n$, where for every a_i, $a_i \in \Sigma$ and there is $x_i \in A$, such that $a_i \preccurlyeq x_i$. $x_1 x_2 \cdots x_n \in A^n$, therefore $x_1 x_2 \cdots x_n \in A^s$. Clearly $dc(\{x_1 x_2 \cdots x_n\}) \subseteq dc(A^s)$, but by fact 5, $dc(\{x_1 x_2 \cdots x_n\}) = dc(\{x_1\}) \cdots dc(\{x_n\})$. So since $a_i \preccurlyeq x_i$ for all $i = 1, \ldots, n$ then $a_i \in dc(\{x_i\})$ for all $i = 1, \ldots, n$, so $x = a_1 a_2 \cdots a_n \in dc(\{x_1 x_2 \cdots x_n\}) \subseteq dc(A^s)$ and we get a contradiction.

Fact 7. If $ax \preccurlyeq by$ and $a \neq b$ then $ax \preccurlyeq y$.

2.4 *Characterization of levels of knowledge*

Now we look at the possibility of characterization of levels of knowledge.

Let $L(A, H) = L(A)$ be a level of knowledge. Let $\bar{L}(A)$ denote the complement of $L(A)$ with respect to Σ^s. Then $\bar{L}(A)$ is upward closed and under each element y of $\bar{L}(A)$ there is a minimal element x. Let $m(A) = \{x_1, \ldots, x_k\}$ be the set of minimal elements of $\bar{L}(A)$. Then the elements of $m(A)$ are mutually incomparable and since $\preccurlyeq$ is a well partial ordering, $m(A)$ is finite. Now we get:

$$\bar{L}(A) = \{y | \exists x \in m(A) \ x \preccurlyeq y\} \quad \text{i.e.} \quad L(A) = \{y | \forall x \in m(A) \ x \not\preccurlyeq y\}$$

Thus the level of A is completely characterized by the finite set $m(A)$ and we get the next theorem.

Theorem 3. *There are only countably many levels of knowledge and in fact all of them are regular subsets of* Σ^s *(or* Σ^s*).*

Proof. Since $m(A)$ is finite, a finite automaton can *clearly* be designed to test whether $x \preccurlyeq y$ holds for some element x of $m(A)$, where y is the input. The fact that there are only countably many levels of knowledge follows immediately. □

COROLLARY 1

The membership problem for a level of knowledge is solvable in linear time.

For a set $L(A)$ we could also look at the set of all maximal elements of $L(A)$ (with respect to $\preccurlyeq$). Clearly these maximal elements are also mutually incomparable. However, the set $L(A)$ need not be characterized by them since if A is common knowledge ($L(A) = \Sigma^s$) then the set of maximal elements is empty. The set of maximal elements is also empty if $L(A) = \Delta^s$ where Δ is a proper subset of Σ. Since distinct sets $L(A)$ can have the same maximal elements, maximal elements cannot characterize $L(A)$.

However, with finite levels, maximal elements do in fact characterize $L(A)$.

Theorem 4. *If L is a non-empty finite subset of Σ^s, then L is downward closed iff for some k and $x_i : i \leqslant k$ where $x_i \in \Sigma^s$,*

$$L = \bigcup_{i=1}^{k} dc(\{x_i\})$$

Proof. Clearly if L is a union of downward closed sets, then by fact 4, L is downward closed.

Conversely, let $L = \{x_1, \ldots, x_m\}$ be a finite downward closed set.
Consider $L' = \cup_{i=1}^{m} dc(\{x_i\})$. Clearly $L \subseteq L'$. Let $x \in L'$. Then there is i such that $x \in dc(\{x_i\})$. Since $x_i \in L$ and L is downward closed, then $x \in L$. So $L = L'$ and therefore L' is the required representation of L. □

Remark. Note that we could have taken just the maximal elements of L and they would be mutually incomparable in that case. If two finite sets $L(A)$, $L(A')$ have the same maximal elements, they must coincide.

In order to analyse infinite levels, we first need to establish some properties of downward closed sets. The following theorem generalizes the representation from the previous theorem to the case of infinite levels. It will be used in obtaining a "normal form" theorem for the levels of knowledge.

DEFINITION 6

A subset L of Σ^s is *star-linear* iff there exist strings $x_1, \ldots, x_{m+1}$ and subsets $\Delta_1, \ldots, \Delta_m$ of Σ such that $L = dc[(\{x_1\})\Delta_1^*(\{x_2\})\Delta_2^* \cdots \Delta_m^*(\{x_{m+1}\})] \cap \Sigma^s$.

Theorem 5. *If L is a subset of Σ^s, then L is downward closed iff L is a finite union of star-linear sets.*

Proof.

$\Leftarrow$

Star-linear sets are downward closed, so by fact 4, so is L.

$\Rightarrow$

L is a downward closed subset of Σ^s, so L is regular and L is the language accepted by some finite automaton M ($L = L(M)$).

Let S be the set of all states of M. Let $s \xrightarrow{x} t$ iff there is a sequence of transitions from the state s to the state t labeled by the symbols forming the string x. We define an equivalence relation $\sim$ on S: $s \sim t$ iff $\exists x, y\ s \xrightarrow{x} t$ and $t \xrightarrow{y} s$. Intuitively, two states are $\sim$-equivalent iff they are in the same loop in M. Now we define a new automaton M', whose states are the equivalence classes $[s]$ of $\sim$. $[s] \xrightarrow{\sigma} [t]$ if there exist $s' \in [s]$ and $t' \in [t]$ such that $s' \xrightarrow{\sigma} t'$. We also add transitions $[s] \xrightarrow{\sigma} [s]$ if $\exists s', s'' \in [s]$ such that $s' \xrightarrow{\sigma} s''$ in M. The accepting states of M' are the equivalence classes of accepting states in M, and the initial state of M' is the equivalence class of the initial state q_1 of M.

It is easy to see that $L(M) \subseteq L(M')$. We will show that $L(M') \subseteq L(M)$.

Let $x \in L(M')$. $x = a_1 a_2 \cdots a_m$. $[s_1] \xrightarrow{a_1} [s_2] \xrightarrow{a_2} [s_3] \cdots [s_m] \xrightarrow{a_m} [s_{m+1}]$ where s_{m+1} is accepting in M and $s_1 = q_1$ is the initial state.

If $[s_1] \xrightarrow{a_1} [s_2]$, there must be states s'_1, s'_2 such that $s'_1 \in [s_1]$, $s'_2 \in [s_2]$ and $s'_1 \xrightarrow{a_1} s'_2$ in M.

Let y_1, y_2 be strings such that $s_1 \xrightarrow{y_1} s'_1$, $s'_2 \xrightarrow{y_2} s_2$, so we have a string $z_1 = y_1 a_1 y_2$ for which $s_1 \xrightarrow{z_1} s_2$ in M. Repeating this procedure we can find strings $z_2, \ldots, z_k$ such that $s_1 \xrightarrow{z_1} s_2 \xrightarrow{z_2} s_3 \cdots s_m \xrightarrow{z_m} s_{m+1}$ and for all $j \leqslant k$, $a_j \preccurlyeq z_j$. Therefore $z = z_1 z_2 \cdots z_m \in L(M)$ (s_{m+1} is accepting in M) and since $L(M)$ is downward closed and $x \preccurlyeq z$, $x \in L(M)$.

We have shown that $L(M) = L(M')$. We show now that the latter has the required form.

The only loops in M' are of the form $[s] \xrightarrow{\sigma} [s]$. A state $[s]$ is *cycling* if there is a σ such that $[s] \xrightarrow{\sigma} [s]$.

Let B_i be a sequence $[s_1] a_1 [s_2] a_2 \cdots [s_{l+1}]$ from the initial state $[s_1] = [q_1]$ of M' to an accepting state $[t] = [s_{l+1}]$, where the a_j are symbols such that $[s_j] \xrightarrow{a_j} [s_{j+1}]$, and $[s_j] \neq [s_{j+p}]$ for any p (there are no *loops* in the sequence). With this sequence we associate a star-linear set $Sim(X) = dc[\Delta_1^s a_1 \Delta_2^s \cdots a_l \Delta_{l+1}^s] \cap \Sigma^s$ where, for all i, if $[s_i]$ is non-cyclic, then $\Delta_i = null$ and otherwise, $\Delta_i = \{\sigma \mid [s_i] \xrightarrow{\sigma} [s_i]\}$. Then the set of non-repeating strings that take us from $[s_1]$ to $[t]$ is exactly X.

If we take the union of all such sets X (there are finitely many of them), then we get the characterization of $L(M')$ in the required form. □

We have found a representation of downward closed sets, and therefore of levels of knowledge. We prove now that there is a unique minimal representation of such a form.

Theorem 6. *Every star-linear set X is directed with respect to the embeddability relation $\preccurlyeq$. In other words: for all $x', x'' \in X$ there is $x \in X$ such that $x' \preccurlyeq x$ and $x'' \preccurlyeq x$.*

Proof. Let $x' = y_1 v_1 \cdots y_m v_m y_{m+1}$ and $x'' = z_1 w_1 \cdots z_m w_m z_{m+1}$ where $y_i, z_i \in dc(\{x_i\})$, and $v_i, w_i \in \Delta_i^s$ for $i = 1, \ldots, m$. Then if we take $x = x_1 v_1 w_1 \cdots x_m v_m w_m x_{m+1}$, then clearly $x' \preccurlyeq x$, $x'' \preccurlyeq x$ and of course $x \in X$. □

Theorem 7. *If $X \subseteq \cup_{j=1}^{l} Y_j$, where X, Y_j are all star-linear, then there exists a j such that $X \subseteq Y_j$.*

Proof. Suppose that $X \subseteq \cup_{j=1}^{l} Y_j$ and for all j, $X \nsubseteq Y_j$, then there are $x_j \in X$, $j = 1, \ldots, l$, such that $x_j \notin Y_j$. By theorem 6 applied $l-1$ times, we can find $x \in X$ such that for all j, $x_j \preccurlyeq x$. Since all Y_j are downward closed, $x_j \preccurlyeq x$ and $x_j \notin Y_j$, so x is not in any Y_j. Thus $x \notin \cup_{j=1}^{l} Y_j$, but $x \in X$ and we get a contradiction. □

Theorem 8. *If L is downward closed then L has a unique representation as a finite minimal union of star-linear sets.*

Proof. Clearly we have a minimal representation of L, $L = \cup_{i=1}^{k} X_i$ and suppose we have some other representation of L of the same form $L = \cup_{j=1}^{l} Y_j$, where all X_i and Y_j's are star-linear sets. Then for every i, $X_i \subseteq \cup_{j=1}^{l} Y_j$. So by theorem 7, for every X_i there is Y_j such that $X_i \subseteq Y_j$. Similarly, by applying theorem 7 to Y_j we get some m such that $Y_j \subseteq X_m$, so we have for every i, some j, m such that $X_i \subseteq Y_j \subseteq X_m$. By minimality of the $\{X_i\}_{i=1,\ldots,k}$ representation $i = m$, so we have $X_i \subseteq Y_j \subseteq X_i$ and hence $X_i = Y_j$. Similarly, every Y_j equals some X_i and the two representations are the same. □

COROLLARY (Normal form theorem)

Every level of knowledge L of a formula A in a distributed system has a unique representation as a finite minimal union of star-linear sets.

2.5 *Representation of levels of knowledge using minimal strings of the complement*

It turns out that the representation of levels of knowledge using star-linear sets is convenient if we want to realize *at least* a given level of knowledge and if we have both synchronous and asynchronous communications in the system. It is not appropriate if we want to realize *at most* a given level and if we have only synchronous communications in the system. In such cases the following theorem is useful:

DEFINITION 7

Given strings $x_1, \ldots, x_k$, not containing repetitions, let $N(x_1, \ldots, x_k)$ be the set $\{y \mid \forall i \leqslant k, x_i \not\leqslant y\}$.

Theorem 9

(a) $N(x_1, \ldots, x_k) = \cap N(x_i): i \leqslant k$.
(b) *If* $x = a_1 \cdots a_m$ *then* $N(x) = (\Sigma - a_1)^s \cdots (\Sigma - a_m)^s$.

Proof. The first part is obvious.

To see (b) we use induction on m.

First of all, it is clear that $N(x)$ includes the right hand side. For suppose $y \in (\Sigma - a_1)^s \cdots (\Sigma - a_m)^s$. If y has no a_1, then it is certainly in $N(x)$. Else the first occurrence of an a_1 in y must be from the $(\Sigma - a_2)^s$ at the earliest. y is then of the form $y_1 a_1 y_2$ where y_1 contains no a_1 and y_2 is in $(\Sigma - a_2)^s \cdots (\Sigma - a_m)^s$. By induction hypothesis $a_2 \cdots a_m \not\leqslant y_2$ and so $x \not\leqslant y$.

Suppose now that y is in $N(x)$. We want to show that $y \in (\Sigma - a_1)^s \cdots (\Sigma - a_m)^s$. If $m = 1$ then this is immediate. Otherwise we know that $x \not\leqslant y$.

If $a_1 \not\leqslant y$, then $y \in (\Sigma - a_1)^s$ and hence $y \in (\Sigma - a_1)^s \cdots (\Sigma - a_m)^s$.

If $a_1 \leqslant y$ then $y = y_1 a_1 y_2$, where $a_1 \not\leqslant y_1$ and $a_2 \cdots a_m \not\leqslant y_2$. In that case $y_2 \in (\Sigma - a_2)^s \cdots (\Sigma - a_m)^s$ by the induction hypothesis and $y_1 \in (\Sigma - a_1)^s$. Since $a_1 \neq a_2$, the result follows. □

3. Model of a distributed system

We assume that there are a finite number of processes, $1, \ldots, n$, which compute and communicate with each other either by asynchronous messages or by broadcasts. Our network is assumed to be fully connected[3] (there is a channel from every process to every other process).

Asynchronous communication consists of two phases: *send* and *receive*. All messages sent are ultimately delivered (and *in the order* in which they were sent) but the delay (transmission time) may be arbitrarily long.

Broadcasts are fully reliable, synchronous communications[4] where all processes involved simultaneously receive the message sent by one of them. Our broadcasts are therefore similar to CSP communication, but allow for more then two particpants at a time. Later on we will point out a limitation of CSP that follows from our results.

We consider three kinds of systems – systems where only asynchronous communication is available, systems where only synchronous communication is available and systems where both kinds of communication are available.

There is assumed to be a global clock in the background which orders all events, but this clock is not assumed to be available to the processes. Time is discrete, the clock is set to 0 at the beginning of every computation and grows in increments of one.

In order for the processors to have something to communicate, we ensure that for every processor in our system there are facts known initially only to this processor. To accomplish that we assume that every process starts its computation with some initial value, a finite string of 0's and 1's. This initial value may correspond to some local non-deterministic input, e.g. the result of a sequence of coin tosses by a processor.

At each moment of global time, zero or more events may take place, at most one at each process. This finite set of local *events* constitutes a *global* event and a *global history* is simply a possible sequence of global events. What global histories are possible is determined by the programs of the various individual processes as well as by the properties of the means of communication. The *protocol* **P** is just the set of all possible global histories, closed under the prefix operation.

Different models of distributed systems have been used by different authors. Our model is similar to that used by Parikh & Ramanujam (1985, pp. 256–268), Halpern & Moses (1990) and by Chandy & Misra (1986).

There are, however, some small differences: we assume that there is no global clock in the system accessible to the processes ("common clock" in Parikh & Ramanujam 1985, pp. 256–268). Parikh & Ramanujam (1985, pp. 256–268) do not have this

---

[3] If the network is not fully connected then some levels of knowledge may be impossible to realize due to the lack of communication capabilities, e.g. if a processor is isolated (cannot communicate with anyone) then the other processes cannot learn anything from that process. Interesting questions arise in case of a directed network where every process may communicate with every other process but some communications are necessarily indirect (go through other processes). We will not analyse this case here.

[4] The two kinds of communications can be thought of as two kinds of communication media e.g. mailing system (asynchronous) and telephone lines (synchronous). Since we allow for synchronous communication between more than two processes at a time, our telephone system must have "conference call" capability. Note that the 'handshake' in CSP can be thought of as a special case of a broadcast involving exactly two processes.

restriction. In their terminology, our local histories are always individual views of "time-free" global histories.

Chandy & Misra (1986) do not allow different events to occur locally in different sites at the same instant of time. We shall not impose this restriction. This can be interpreted in two ways: either we can treat time in our system as "less refined" than in Chandy & Misra (1986), or we can interpret our system as synchronous, events in all sites being governed by the same global clock. Differences in local time will then arise solely from the fact that a process may not have an event happening at every moment of global time. If a process is inactive in a certain round (its action is the "null" event), it cannot perceive that time has passed. Hence even while assuming that our system runs in synchronous rounds, we ensure that processes are not able to record global time. Consequently they cannot draw any conclusions about the others merely by observing their own local clock.

There are some terminological differences between our description of a distributed system and that of Halpern & Moses. Following Parikh & Ramanujam (1985, pp. 256–268), we shall denote the set of all global histories (runs in Halpern & Moses 1990) as the protocol. Halpern & Moses (1990) use the word protocol to describe the rules governing actions of processors (effectively generating the set of runs).

3.1 *Definitions*

Here we formally specify our class of models. Let $N=\{1,\dots,n\}$ be the set of all processors. Every processor i has infinitely many possible initial states v and an initial state is a string of 0's and 1's ($v\in\{0,1\}^*$). We denote the set of initial states for i by V_i. The set of global initial states is $\mathscr{V}=\Pi_{i=1}^{n} V_i$.

From now on we will use lower case letters to denote everything pertaining to a single process. Capitals without subscripts will be used where all the processes are involved (e.g. v_i is an initial state of a processor i, while V is an initial configuration of the whole system: $V=(v_1,\dots,v_n)$). $\mathscr{V}$ is the set of all such V. v_i, initial input of the processor i provides interpretation for private facts in our logic, i.e. $Q_{i,m}$ is true iff the mth bit of v_i is 1. The mth private fact $P_{i,m}$ of a processor i is whether $Q_{i,m}$ or its negation is true.

Events. E_i denotes the set of all events in which processor i can participate (events *local* to i). There are the following types of events (or actions):

(1) L_i: Local computation steps. (We assume that for $i\neq j$, $L_i\cap L_j=\phi$)
(2) $s(i,j,m)$: Sending a message m to a processor j, $j\in N$.
(3) $r(i,j,m)$: Receiving a message m from a processor j, $j\in N$.
(4) $bc(i,U,m)$: Sending a broadcast m to a group of processors $U, i\in U\subseteq N$. The same event is also in E_j for all j in U.

$$E_i = L_i\cup\{s(i,j,m)\mid m\in M, j\in N\}\cup\{r(i,j,m)\mid m\in M, j\in N\}$$
$$\cup\{bc(j,U,m)\mid m\in M, i,j\in U\subseteq N\}\cup\{bc(i,U,m)\mid m\in M, i\in U\subseteq N\}$$

Note that the last two items can be combined into one as $\{bc(j,U,m)\mid m\in M, i\in U\subseteq N\}$. M is the set of messages, defined below.

We define the set of *global events* $\mathbf{G}$ in our system. $\mathbf{G}\subseteq\Pi_{i=1}^{n}(E_i\cup\{\text{null}\})$ (a cartesian product) such that if $(e_1,\dots,e_i,\dots,e_n)\in\mathbf{G}$ for some i and $e_i=bc(j,U,m)$ then for all $i' \in U$, $e_{i'} = bc(j, U, m)$. If $e_i = \text{null}$ for some i, it means that there is no local event at i at this point. Note that null is not local to any process. We use the notation $(G)_i$ to denote the ith coordinate of G, so $(e_1, \dots, e_i, \dots, e_n)_i = e_i, \dots, e_i$.

Histories. A history (a run) is an input value followed by a sequence of events. The set of all possible histories of the system will be called the protocol **P**. So $\mathbf{P} \subseteq \mathscr{V}; \mathbf{G}^*$. Protocols are always closed under taking an initial segment of a history: $H \in \mathbf{P}$ implies that every H' which is an initial segment of H is in **P**.

We will require that for every receive in every history in every protocol, there is exactly one corresponding send, which occurs before the receive (this condition will be called *time-consistency*).

We say that two histories H and H' are *compatible* iff they start with the same input values (initial states).

We define the *concatenation* of compatible histories:

If $H_1 = V; G_1; \dots; G_k$, and $H_2 = V; G'_1; \dots; G'_l$, then the concatenation of H_1 and H_2 is the history $H = V; G_1; \dots; G_k; G'_1; \dots; G'_l$.

Local histories are the projections of global histories onto the sets of local events of the processors. They are "time-forgetting", i.e. they erase null events.

We assume that a global event – the ticking of the clock – takes place even if no local events take place at a particular moment (this corresponds to events of the form (null, …, null)), and so the length of a global history is just the amount of time elapsed. However, what each process sees at any moment of time is its local event, if any, and its local history is simply the sequence of local events. Given i, and the global history H, the local history h_i is uniquely defined and we let Φ_i be the map which takes us from H to h_i.

We can inductively define Φ_i's as follows:

$$\Phi_i((v_1, \dots, v_n)) = (v_1, \dots, v_n)_i = v_i,$$

$$\Phi_i(H; G) = \begin{cases} \Phi_i(H); e_i, & \text{iff } (G)_i = e_i \in E_i, \\ \Phi_i(H), & \text{iff } (G)_i = \text{null}. \end{cases}$$

So Φ_i is like $(\cdot)_i$ (when we extend the projection operation from events to histories) except that it erases null events.

Its local history is all that a processor sees, so all global histories which correspond to the same local history h_i look the same to the processor i. Note that the length of $\Phi_i(H)$ is less than or equal to the length of H. In fact *length* $(\Phi_i(H)) = \textit{length}(H)$ iff there are no *null* events on i in H.

For every i we can define an equivalence relation on the set of global histories:

$$H \approx_i H' \text{ iff } \Phi_i(H) = \Phi_i(H').$$

If U is a subset of N, then we let $\approx_U^*$ be the reflexive transitive closure of $\cup \approx_i : i \in U$.

We use capital letters to denote global histories, events etc., lower case letters to denote local histories, events etc.

Time. The *global time* of an event G in a global history H, *Time* (G, H), is the length of the initial segment of H up to and including G, the time when the event G has occurred in the history H.

Note that since null $\notin E_i$ for all i, processes do not have access to the global clock.

They can introduce their own local logical clocks, but these clocks do not have to coincide. *Local time* of e_i in h_i can be defined as *time* (e_i, h_i) – the length of h_i up to and including e_i.

The lack of the access to the global clock, together with the closure conditions for the protocol guarantee that the only possible "causality" ordering that can be defined corresponds in case of asynchronous systems to Lamport's (1978) "happened before" ordering[5].

Messages. Messages m are knowledge formulae (formulae of the form xA where x is a string of knowledge operators $K_{i_1}K_{i_2}\cdots K_{i_n}$ and A is some boolean combination of $P_1, \ldots, P_n$). We assume that the processes are "honest", they only send (or broadcast) messages which they know are true. Formally, if $\Phi_i(H) = h_i; s(i,j,B)$ or $\Phi_i(H) = h_i; bc(i,U,B)$ and $H \in \mathbf{P}$, then if $\Phi_i(H') = h_i$ then $H' \models K_i B$. Later on we will also allow certain 'common knowledge' symbols C_U to occur in addition to the K_i.

We remark that the notion of a message and that of a history should really be defined via mutual recursion since the truth of a message depends on the histories that are possible and the histories that are possible can contain only true messages. However, for the messages of the type we have described, the truth of a message depends only on the portion of the history that has already elapsed. Hence the recursion is allowable. If the messages were to contain future modalities, then there could be problems, which arise due to circularity.

We illustrate our semantics by means of an example. Suppose that for $i \leqslant 3$, P_i is a private fact of process i. Suppose also that in a history H, P_i are all true. Now process 3 receives a message from process 2 that $P_1 \to P_3$. 3 knows that the message is correct because P_3 is in fact true. However, 3 has never sent any messages to anyone at all. Now 3 can reason, "2 cannot know that P_3 is true so 2 must know that P_1 is false." We show how to do this argument in our framework. If we replace H by H' where P_3 is false, but otherwise the same as H, then $H \approx_2 H'$. Now 2 must have sent in H only a message that he knew to be true, so we must also have $H' \models K_2(P_1 \to P_3)$. Hence $H' \models (P_1 \to P_3)$ so that $H' \models \neg P_1$. Since we only used the fact that $H \approx_2 H'$, we get $H \models K_2(\neg P_1)$. However, this argument could be applied to any $H'' \approx_3 H$ so that we get $H \models K_3(K_2(\neg P_1))$.

Closure conditions for the protocol. We impose some additional conditions on the protocol $\mathbf{P}$. We want to ensure that the initial state of i, (v_i) cannot be known to any other process j at any run of the system, unless j *learns* about v_i from some communication. We want to exclude the possibility that something may be known "accidentally". To achieve that we will make sure that all initial states are possible. Moreover, if v_i is the initial state of i, all other strings v_i' will remain possible for j as initial states of i, unless j gets some message to the contrary (directly from i or via some other processors).

(1) All vectors of input values are possible: $\forall V$ such that $V = (v_1, \ldots, v_n)$ where every v_i is a sequence of 0's and 1's, there is some $H \in \mathbf{P}$ such that for some H', $H = V; H'$ (note that H' cannot itself be a history since it lacks an input value).

[5] $e_1 \to e_2$ iff e_1 is send, e_2 is receive of the same message or e_1, e_2 are local to the same process and e_1 occurred earlier than e_2. e_1 happened before e_2 iff $e_1 \to^* e_2$ where $\to^*$ is the reflexive, transitive closure of $\to$. In a system where we allow broadcast one more condition is needed in the definition of $\to$. $e_1 \to e_2$ if e_1 and e_2 are two local projections of the same broadcast.

(2) No sequence of local events on some group of processes can influence possible actions of some other group of processes unless there are some communications (assuming that both groups are disjoint).

For that we need some closure conditions on the set of all protocols. The first condition we use is due to Chandy & Misra (1986) (it is the first of their *principles of computation extension*).

We need one definition:

Let $G = (e_1, \ldots, e_n)$, G is *on* U if $U = \{i | (G)_i \neq \mathit{null}\}$ (so U is the set of processes which have some local events in G).

Closure conditions:

(i) *Extension rule*: Suppose that $\forall i \in U$, $H \approx_i H'$, G is on U, and none of $(G)_i$ is a receive $r(i,j,m)$ for any j not in U. Then

$$H' \in \mathbf{P}, \quad H; G \in \mathbf{P} \Rightarrow H'; G \in \mathbf{P}.$$

The extension rule guarantees that if we have a protocol $\mathbf{P}$, some history H in $\mathbf{P}$ and some action of a group of processes U is possible in H, then the same action must be possible in every history H' which looks the same to all processes in U unless it violates time-consistency. In order to see why e_i cannot be allowed to be a receive from a processor outside U, let us look at an example:

$$\text{Let } N = \{1, 2, 3\}, \quad U = \{1, 2\}.$$

$H = V; (\mathit{null}, \mathit{null}, s(3, 1, m))$, $H' = V; (\mathit{null}, \mathit{null}, \mathit{null})$ for some input V. Clearly $H \approx_1 H'$ and $H \approx_2 H'$. If we take $G = (r(1, 3, m), \mathit{null}, \mathit{null})$ such that $H; G \in \mathbf{P}$ then requiring $H'; G$ to be in $\mathbf{P}$ would violate time-consistency.

The following conditions assure that no process can get any additional information about the other processes by observing its own local events (no hidden synchronization). These conditions are necessary because (unlike Chandy & Misra 1986) we allow local events at different sites at the same instant of time. Condition (ii) says that if some local events have occurred in parallel, and the sets of participating processes were disjoint, they could have occurred in sequence. We'll call it the splitting rule.

(ii) *Splitting rule*: $G = (e_1, \ldots, e_n)$, $G \notin V$, G is on U. Given U_1, U_2 such that $U_1 \cup U_2 = U$ and U_1, U_2 disjoint, then we can "split" G into G_1 and G_2:

$$H; G \in \mathbf{P} \Rightarrow H; G_1; G_2 \in \mathbf{P},$$

where $(G)_i = (G_1)_i$ for $i \in U_1$, $(G)_i = (G_2)_i$ for $i \in U_2$, $(G_1)_j = \text{null} = (G_2)_k$ for $j \notin U_1$, $k \notin U_2$ provided that we don't split any broadcasts: $(G)_i = bc(i, V, m) \rightarrow V \subseteq U_1 \vee V \subseteq U_2$.

Condition (iii) says that if some local events have occurred in sequence, the sets of participating processes were disjoint, and there was no send receive pair in them, they could have occurred in parallel.

(iii) *Joining rule*: Given U_1, U_2 such that $U_1 \cup U_2 = U$ and U_1, U_2 disjoint, let G_1 be on U_1, G_2 on U_2, and if there are no i, j such that $(G_1)_i = s(i, j, m)$ and $(G_2)_j = r(j, i, m)$.

$$H; G_1; G_2 \in \mathbf{P} \Rightarrow H; G \in \mathbf{P},$$

where $(G)_i = (G_1)_i$ for $i \in U_1$, $(G)_i = (G_2)_i$ for $i \in U_2$.

Systems. We consider three kinds of systems. *Asynchronous* systems are the systems as described above but without broadcasts. The only communications are via send and receive. *Synchronous* systems are systems in which all communications are done using broadcasts and we *do not* have the events send and receive. Finally, we use the name *mixed communications systems* for systems with both kinds of communications available.

4. Realization of levels of knowledge in distributed systems

Since we know now that every level of knowledge is a downward closed set of strings, we can ask whether given a downward closed set of strings L we can find some formula A and some run of a distributed system (history H) such that $L = L(A, H)$. The answer, it turns out, depends on the kind of communication available in the system. In a system with unreliable delivery time (asynchronous system) we are able to realize only finite levels of knowledge (this generalizes the result of Halpern & Moses (1990) that no common knowledge can be achieved in an asynchronous system). In systems where all communications are instantaneous broadcasts with at least 3 processors – synchronous systems – all levels of knowledge can be realized. If there are only 2 processors, and broadcasts are the only medium of communication, then the finite levels containing strings longer then 1 cannot be realized. In the full systems, where there are two communication media: synchronous broadcast, and asynchronous send and receive, all levels can be realized.

So in the following sections we separately analyse these three kinds of systems:

(1) Systems where only asynchronous communications are available.
(2) Systems where only synchronous communications are available.
(3) Systems where both synchronous and asynchronous communications are available.

Before we proceed with realizing levels of knowledge, we say something about the properties of the formulae, which we will look at.

DEFINITION 8

A formula A is *persistent* if whenever $H \models A$ and H' extends H, then $H' \models A$.

Theorem 10. *If A is persistent then so is $K_i(A)$ for any i.*

Proof. Suppose $H \models K_i(A)$ and H' extends H. Suppose that for some H'', $\Phi_i(H'')$ equals $\Phi_i(H')$. Then for some initial segment H_t of H'', $\Phi_i(H_t)$ equals $\Phi_i(H)$. Hence H_t satisfies A and by the persistence of A, so does H''. Since H'' was arbitrary with $\Phi_i(H'') = \Phi_i(H')$, H' satisfies $K_i(A)$. □

Theorem 11. *Every formula A which is a boolean combination of P_i's is persistent.* □

COROLLARY 1

Every formula of the form xA, where A is a boolean combination of P_i's, and x is a string of knowledge operators, is persistent. □

From now on we will look only at persistent formulae of the form as in corollary 1 after theorem 11.

4.1 *Asynchronous case*

We now look at the question of the levels of knowledge in asynchronous systems. The only possible communications are via send and receive, where the arrival time of the message is not guaranteed. In the formal model we exclude broadcasts, so no event is local to more then one processor.

What are the consequences of the fact that messages, although eventually delivered, may remain in the mail for an unbounded amount of time? Suppose that a process i sends a message to a process j that a certain formula A (whose truth value is invariant over time) is true. Then when process j receives the message, it knows A and that i knows A. Thus $K_j(K_i(A))$ is true. However, i does not know that j has received the message and hence $K_i(K_j(K_i(A)))$ does not hold until i receives an acknowledgement from j.

Let's show that this fact is valid in our model. We'll take only a 2 processor system, the processors are 1 and 2. Let I be an initial configuration of the system in which the first bit of an input of the first processor is 1 (and this single bit is the whole input of 1). So $I=(v_1, v_2)$, $v_1=1$. Let P_1 say that the input value of the processor is 1, so $P_1=Q_{1,1}$. Slightly abusing notation we will write that $I \models P_1$, instead of $I \models P_1 = Q_{1,1}$. P_1 is private to 1. Now let H be a history which starts in the initial configuration I and in which 1 sends a message to 2 informing him that P_1 and this message is received by 2 in the next instant of time. $H=I;\ (s(1,2,P_1), \mathit{null});\ (\mathit{null}, r(2,1,P_1))$, and $H \models K_2K_1P_1$.

Let H' be a history in which the same message is sent, but it is delayed, it is not received by 2 in the second instant of time: $H'=I;\ (s(1,2,P_1), \mathit{null});\ (\mathit{null}, \mathit{null})$.
Because of the extension rule H' is in the protocol.
Clearly $H \approx_1 H'$.
Let H'' be a history with the same input configuration, but in which 1 is slow and hasn't sent anything yet: $H''=I;\ (\mathit{null}, \mathit{null});\ (\mathit{null}, \mathit{null})$ (again, because of the extension rule, H'' is in the protocol).
$H'' \approx_2 H'$.
Let $H'''=I_1$; where I_1 is an initial configuration in which 2 receives the same input as in H, but 1's input is 0 instead of 1. H''' is in the protocol because of the first closure condition (all inputs are possible).
$H''' \approx_2 H''$, and we get:
$H''' \approx_2 H'' \approx_2 H' \approx_1 H$, so $H''' \approx_2 H' \approx_1 H$. Since $H''' \not\models P_1$ then $H \not\models K_1K_2P_1$, hence $H \not\models K_1K_2K_1P_1$.

It seems reasonable to suppose that a back and forth interchange of messages will only make a finite amount of difference and we proceed to show now that this is indeed the case.

Theorem 12. *Let A be a formula of the form xB, where x is a string of knowledge operators and B is a boolean combination of private facts P_i. Let $H \not\models K_jA$. Then if $H''=H;H'$ and $H'' \models K_jA$, then there is a receive in $\Phi_j(H')$ (either of the form of $r(j,l,m)$ for some l, or $bc(l,U,m)$ where $j \in U$, $l \neq j$). Informally: a process may learn something about the others only when it receives a message.*

Proof. Proof by induction on the length of H'. If the length of H' is 0 then the theorem clearly holds. So suppose that H' is $H'_0; G$. If $H; H'_0 \nvDash K_j A$ then there is some $H_1 \approx_j H; H'_0$ such that $H_1 \nvDash A$. By the splitting rule if G is not a broadcast, we can split it: there are G_2, G_1 such that $H; H'_0; G_2; G_1$ is in $\mathbf{P}$ where $(G_1)_j = (G)_j$, $(G_1)_i = \text{null}$ for $i \neq j$; $(G_2)_i = (G)_i$, for $i \neq j$, $(G_2)_j = null$. $H; H'_0; G_2 \approx_j H; H'_0$, so $H_1 \approx_j H; H'_0; G_2$. By the extension rule $H_1; G_1$ must be in $\mathbf{P}$ (G_1 was not a receive!). Furthermore $H_1; G_1 \approx_j H; H'$. If x was empty (A is a boolean combination of private facts), then if $H_1 \nvDash A$, then $H_1; G_1 \nvDash A$. If x is $K_s y$, then by the induction hypothesis since s has not received any message in G_1 he didn't learn anything, so if $H_1 \nvDash A$, then $H_1; G_1 \nvDash A$. So in any case $H; H' = H'' \nvDash K_j A$. □

As a consequence we get the following theorem, which is essentially Chandy & Misra's (1986) theorem 5.

Theorem 13. (Chandy & Misra 1986) *If for some histories H, H' such that H is an initial segment of H':*

$$H' \vDash K_{i(1)} K_{i(2)} \cdots K_{i(p)} A \text{ and } H \nvDash K_{i(p)} A$$

then in $H' - H$ there must be a sequence of messages: $m_{p-1}, m_{p-2}, \ldots, m_1$ such that m_{p-1} is sent by $i(p)$ and reaches $i(p-1)$ (maybe via some other processes), $\ldots, m_{i(1)}$ is sent by $i(2)$ and (maybe indirectly) reaches $i(1)$ (the messages may be different but they must all imply A). Moreover if A doesn't depend on any local event of $i(p)$ (its truth value depends on some event $e \notin E_{i(p)}$) then there must be some event of the form $r(i(p), l, m)$ occurring after H but before $s(i(p), i(p-1), m_{p-1})$.

Proof. By induction on p using the previous theorem.

Theorem 14. *Suppose communication is asynchronous and A is a persistent formula. Let H and H' be global histories such that H' extends H. Then $L(A, H')$ includes $L(A, H)$ and there is a finite X such that $L(A, H') \subseteq dc(X; L(A, H))$.*

Proof. The fact that the level increases with time follows from the fact that we deal with persistent formulae.

For the second part we use the theorem of Chandy & Misra (1986).

Since there are only finitely many events G between H and H', there are only finitely many possible sequences of events as above and a finite set of strings x for which the conditions of theorem 13 are satisfied. Let X be the set of these strings, then X satisfies the conditions of the theorem. □

Now we characterize precisely how the level of knowledge grows when a message is received.

Theorem 15. *Suppose that a history H is exactly the following sequence of messages:*

$$s(i_m, i_{m-1}, R_m); \quad r(i_{m-1}, i_m, R_m)$$
$$s(i_{m-1}, i_{m-2}, K_{i(m)} R_m); \quad r(i_{m-2}, i_{m-1}, K_{i(m)} R_m) \ldots$$
$$s(i_2, i_1, K_{i(3)} \cdots K_{i(m)} R_m); \quad r(i_1, i_2, K_{i(3)} \cdots K_{i(m)} R_m)$$

(For simplicity we have left out the null events of processors), where R_m is initially (in the empty history) known only to i_m; then

$$L(R_m, H) = dc\{K_{i(1)}K_{i(2)}\cdots K_{i(m)}\}.$$

Proof. It is easy to notice that $dc\{K_{i(1)}K_{i(2)}\cdots K_{i(m)}\} \subseteq L(R_m, H)$. The other inclusion we prove by induction on the length of H. If it is 0 then H is empty so R_m is known only to i_m. So assume that the theorem is true for histories of length up to $m-1$.

Let us assume then that there is some y such that $H \vDash yR_m$ and

$$y \notin dc\{K_{i(1)}K_{i(2)}\cdots K_{i(m)}\}.$$

So y must be non-empty. Let then $y = K_l y'$. If l was not mentioned in the sequence of messages (l was not any of $\{i_1, \ldots, i_m\}$ – then l would have known R_m initially (it hasn't received any messages, see theorem 12) – which contradicts our assumption. So it must be that $l \in \{i_1, \ldots, i_m\}$. Then we can express $K_{i(1)}K_{i(2)}\cdots K_{i(m)}$ as $zK_l z'$ where the K_l picked is the leftmost occurrence of K_l in $K_{i(1)}K_{i(2)}\cdots K_{i(m)}$. Let's define H' as the initial segment of H up to the last receive by l. l doesn't learn anything in $H - H'$ (he doesn't receive any messages), so if $H \vDash K_l y' R_m$ then $H' \vDash K_l y' R_m$. $H' = H''; G$ where $(G)_i = r(l, s, m)$. By the splitting rule we can split G into G_1 and G_2, where G_1 is a receive on l, G_2 is *null* on l. $H''; G_1; G_2$ must be in the protocol. $H' \approx_l H''; G_1$, so $H'' \vDash y' R_m$. Then by the induction hypothesis $y' \in dc(\{z'\})$, so $y' \preccurlyeq z'$ and therefore $y \preccurlyeq K_{i(1)}K_{i(2)}\cdots K_{i(m)}$. □

Theorem 16. (a) *Suppose H realizes $L(A, H)$ (for some formula A which is a boolean combination of P_i) and $H \leqslant H'$*[6]*, i.e. all messages in H occur in H' and in the same order, then H' realizes at least $L(A, H)$.*
(b) *Suppose H realizes $L(A, H)$* and *H' realizes $L(A, H')$ Suppose that H'' is the concatenation of H and H'. Then:*

$$L(A, H'') = L(A, H) \cup L(A, H')$$

Proof. Part (a) follows easily by induction on the length of H using the fact that A is persistent.

To see part (b), clearly $L(A, H) \cup L(A, H') \subseteq L(A, H'')$.

Suppose $H'' \vDash yA$. If y is empty, then we are done, since $H \vDash A$.

Otherwise let yA be true in H'', where y is (say) $K_1 y'$. Suppose that $K_1 y' \notin L(A, H)$. Then there must be some point T such that $H; H_T \vDash K_1 y' A, H; H_{T'} \nvDash K_1 y' A$ (for all $H_{T'}$ such that $H_{T'} \leqslant H_T \leqslant H'$). By theorem 12 H_T has as its last event some G such that $(G)_1 = r(1, l, zA)$. So $y' \preccurlyeq K_l z$. But the event G was a part of H'. Message sent and received in H' must have been true in H'. So $H_T \vDash K_1 y' A$, so $H' \vDash yA$. □

Note that H and H' could be executed in parallel. In fact we can take any minimal H'' such that both H and H' are embeddable in H''.

We now show that *all* finite downward closed sets are actually attainable as knowledge levels $L(A)$ of formulas in asynchronous systems.

Theorem 17. *Every finite downward closed set is the set $L(A, H)$ for an appropriate A and H in some asynchronous protocol.*

[6] $\leqslant$ is here embeddability relation defined for histories.

Construction. Let X be a finite downward closed set of strings from Σ^s. We construct a formula A and a history H such that $L(A,H)=X$.

(1) If X is empty then A is the formula *false*.

(2) If X consists of the empty string, then A is the conjunction P_1 and P_2 where: P_1 is a predicate whose truth value is initially known only to process 1, P_2 is a predicate whose truth value is initially known only to process 2, and H is the empty history.

(3) Otherwise X has nonempty strings and therefore the set of all maximal strings M in X is nonempty. Let $M=x_1,x_2,\dots,x_r$. Let $x_i=K_{i(k_i)}\cdots K_{i(2)}K_{i(1)}$ for $i=1,\dots,r$. Then we take A to be a $\bigvee_{i=1}^{r} P_{i(1)}$.

Let

$$\begin{aligned}H_i = {} & s(i(1),i(2),A); r(i(2),i(1),A); s(i(2),i(3),K_{i(1)}A);\\ & \cdots s(i(k-1),i(k),K_{i(k-2)}\cdots K_{i(2)}K_{i(1)}A);\\ & r(i(k),i(k-1),K_{i(k-2)}\cdots K_{i(2)}K_{i(1)}A).\end{aligned}$$

Clearly $L(A,H_i)=dc(x_i)$. Now by theorem 16 if $H=H_1;H_2;\dots H_r$ then $L(A,H)=\bigcup_{i=1}^{r} dc(x_i)$.

Note that H could be any permutation of H_i's. In fact all H_i could be executed in parallel. □

4.2 *Synchronous case*

In this section we will augment the alphabet Σ to a larger alphabet Σ_C which includes symbols C_U where $U\subseteq N$. Semantically, the symbol C_U denotes the set $\{K_i|i\in U\}^s$ and will be referred to as the *common knowledge* of processors in U. However, the element C_U of Σ_C is not the infinite *set of all strings* in the denotation of C_U, but the string C_U where U is explicitly enumerated. The C_U as part of a message is a finite string which denotes an infinite regular set of strings.

Levels of knowledge will continue to be identified with subsets of Σ^s. Thus, for example, the string $C_{1,2}C_{2,3}$ denotes, as a subset of Σ^s, the set of all simple strings consisting of any number of K_1 and K_2 followed by any number of K_2 and K_3.

We start with a preliminary result.

Theorem 18. *The set $L(A)$ is infinite iff it includes common knowledge of A between two distinct processes i and j.*

Proof. One direction is clear, as common knowledge between i and j includes all strings in $\{K_i,K_j\}^s$.

Conversely, suppose that $L(A)$ does not include common knowledge of any two distinct i,j. Then, since $L(A)$ is downward closed, for any such i,j, there must be a maximum $Alt(i,j)$ number of alternations between K_i and K_j in any string in $L(A)$. Let m_A be the largest of these $Alt(i,j)$ and let p be m_A*n^2 (n is the number of processors). Now any nonrepeating string of length greater than $p+1$ has at least $p+1$ alternations, and hence more than m_A for some *specific* alternation between *some* K_i and K_j. Thus no string in $L(A)$ can have length greater then $p+1$ and $L(A)$ is finite. □

In the following theorem we recall that C_U stands semantically for the set $\{K_i|i\in U\}^s$.

Theorem 19. *If we broadcast "xA" where x is a string of knowledge operators and A is a boolean combination of propositions P_i among the group of processes U, then the created level of knowledge of A increases by the downward closure of $\{C_U\}$. Formally: If $H = H'; G$, $\forall j \in U$ $(G)_j = bc(i, U, xA)$, $\forall j' \notin U$ $(G)_{j'} = null$, then:*

$$L(A, H) = L(A, H') \cup dc(\{C_U\})dc(\{x\}).$$

Proof. Let H be as in the premises of this theorem. Then $H \models A$ (our processes are honest and A may be broadcasted). Moreover since it is a property of all histories in the protocol, it is common knowledge that formulae sent (or broadcasted) are true and remain true (all formulae persistent):

$$\models C_N((H = H'; G; H'' \text{ and } (G)_j = bc(i, U, xA)) \rightarrow H \models xA),$$

this implies that for every $U \subseteq N$:

$$\models C_U((H = H'; G; H'' \text{ and } (G)_j = bc(i, U, xA)) \rightarrow H \models xA),$$

so in order to prove that $H \models C_U xA$ it is enough (since common knowledge is closed under *modus ponens*) to show that $H \models C_U(H = H'; G; H'')$ and $(G)_j = bc(i, U, xA)$. But it is a property of **P** that if for some $j \in U$, $(G)_j = bc(i, U, B)$ then for all $j \in U$, $(G)_j = bc(i, U, B)$. So $\forall H_0 \approx_{U^*} H (H = H'; G; H''$ and $(G)_j = bc(i, U, xA)) \rightarrow (H_0 = H_0'; G; H_0''$ and $(G)_j = bc(i, U, xA))$. Therefore $H \models C_U xA$ so $H \models dc(\{C_U\})dc(\{x\})A$, $H \models dc(\{C_U x\})A$. The theorem follows since no process outside of U received any message. □

We now prove that in the presence of synchronous communication, if there are at least three processors, then every downward closed set of strings without repetitions is the level of knowledge of some formula under some history. We have proved that if $m = (x_1, \ldots, x_k)$ is the set of minimal elements of the complement of some downward closed set L, then

$$L = \cap N(x_i) : i \leqslant k.$$

$$\text{If } x = a_1 \cdots a_m \text{ then } N(x) = (\Sigma - a_1)^s \cdots (\Sigma - a_m)^s.$$

The next theorem will show us how to realize $N(x)$. We take a sequence of broadcasts to $N - 1$ processes at a time. Let's assume that at time T, group U_T receives a broadcast and at time $T + 1$ group U_{T+1} (both of $N - 1$ processors). Then the processor sending broadcast at $T + 1$ (a_{T+1}) will be one of the processors in $U_{T+1} \cap U_T$, and the message sent will be the string of common knowledge operators $C_{U_T} \cdots C_{U_1}$ followed by a fixed formula P private to one of the processes in $(\Sigma - a_m)$.

Theorem 20. *Let H be a history in which the private information of s_1 is P_s, and every event G^T occurring in H at time T for $T = 1, \ldots, l$, (where $l = length(H)$) is of the form: $(G^T)_i = bc(a_{s_T}, N - \{a_T\}, m_T)$ for all $a_{s_T} \neq a_T, a_{s_T} \neq a_{T-1}$, $(G^T)_i = null$ otherwise. Where $m_1 = P_{s_1}, m_{T+1} = C_{N - \{a_T\}} m_T$.* Then:

$$L(P_{s_1}, H) = dc(C_{N - \{a_l\}} \cdots C_{N - \{a_1\}}).$$

Proof. Induction on the length of H. Clearly if $l = 1$ then $L(P_{s_1}, H) = C_{N - \{a_1\}}$.

Suppose that for histories up to the length $l - 1$ theorem is true. Now we can use theorem 19: if $H = H'; G^l$ where $(G^l)_i = bc(a_{s_l}, N - \{a_l\}, m_l)$ then $L(P_{s_1}, H) = L(P_{s_1}, H') \cup dc(C_{N-\{a_l\}} C_{N-\{a_l\}} \cdots C_{N-\{a_1\}})$.

Since by induction hypothesis (IH) $L(P_{s_1}, H') = dc(C_{N-\{a_{l-1}\}} \cdots C_{N-\{a_1\}})$ then

$$L(P_{s_1}, H) = dc(C_{N-\{a_l\}} \cdots C_{N-\{a_1\}}). \qquad \square$$

COROLLARY 1

In a system with at least 3 processors, for every x in Σ^s, there exists a history H and a formula A such that $L(A, H)$ is just the set of strings without repetitions in $N(x)$.

Theorem 21. *Every downward closed set L* of strings without repetitions is $L(A, H)$ *for suitable A and H in a synchronous system with at least 3 processors.*

Proof. Let $\{x_1, \ldots, x_k\}$ be the set of minimal elements of the set of strings without repetitions which are not in L. Then L is $N(x_1, \ldots, x_k)$.

If $k = 0$ then all strings without repetitions are in L and L is just common knowledge, which can be achieved by taking the formula A to be a tautology.

Otherwise for each x_j we can find a P_j and a history H_j such that $L(P_j, H_j)$ is exactly $N(x_j)$. Now let H be $H_1; \ldots; H_k$ and A be the conjunction of the P_j. (We assume the P_j are all independent so that if $i \neq j$ then H_i conveys no information about P_j.) Then $y \in L(A, H)$ iff $H \models yA$ iff for all j, $H \models yP_j$ iff for all j, $H_j \models yP_j$ iff for all $j, y \in N(x_j)$ iff $y \in L$. $\square$

Theorem 22. *In a two processor system with only synchronous communication available, no finite level containing strings of length $\geqslant 2$ can be achieved for any formula A.*

Proof. Suppose that $H \models K_1 K_2 A$. If in the empty history 1 knows that 2 knows A, then A must be true in all histories (A persistent) and therefore must be common knowledge. Otherwise 1 must have learned that $K_2 A$ in H, so there must have been a communication between 1 and 2 to that effect, but there are only synchronous communications in the system, and these create common knowledge. $\square$

4.3 *Mixed case – both kinds of communication available*

If we have both kinds of communications in the system, we can directly realize every level. We show first how to realize a level in the normal form. Later we will also show how to realize the level given by the set of minimal strings of the complement. As we will see from the examples in a following section, if we have some specification of the level of knowledge to be achieved, which is incomplete (doesn't specify a unique level), then the two constructions will give us different levels of knowledge (maximal and minimal) satisfying requirements.

Construction

$$L = \bigcup_{i=1}^{k} dc(X_i),$$

where X_i is star-linear.

We take $A = \vee_{i=1}^{k} P_{s(i)}$ where $s(i) \in U_j$, such that $x_{i(1)} = C_{U_j}$. We create H_i to realize the level of knowledge $dc(X_i)$ of a formula $P_{s(i)}$ for every i, and then we take H to be the concatenation of all H_i's.

We can also construct a level specified by a set of minimal strings of the complement.

Construction.

$$L = N(x_1, \ldots, x_k),$$

where $x_i \in \Sigma^s$.

We take $A = \wedge_{i=1}^{k} P_{s(i)}$ where $s(i) \in U_j$, such that $x_i = yC_{U_j}$. We create H_i to realize the level of knowledge $N(x_i)$ of a formula $P_{s(i)}$ for every i, then we take H to be a concatenation of all H_i's.

Let $x_i = C_{U_m} \cdots C_{U_1}$.

(i) $s(i)$ initially sends a broadcast $P_{s(i)}$ to all the processes *not* in U_1.

(ii) Sender of the last broadcast (to $N - U_j$) sends asynchronously all that he knows about $P_{s(i)}$ to one of the processes *not* in U_{j+1} (if there was no broadcast yet, $s(i)$ is a sender).

(iii) Recipient of the last asynchronous message broadcasts all that he knows about $P_{s(i)}$ to all processes in $N - U_{j+1}$. □

4.4 *Limited n-casts*

If in our system we have only a limited broadcast capability, then not all levels of knowledge can be achieved. Precisely:

Theorem 23. *If in a system of N processes, all the broadcasts are to groups of n processes ($n < N$), then no $C_U A$ can be realized for any U with $|U| > n$, for any formula A which is not true in all the histories.*

Proof. Broadcast $bc(i, U, m)$ creates common knowledge of m among U. A sequence of broadcasts of up to n processes (say to $U_1, \ldots, U_k$) will create $dc(C_{U_k} C_{U_{k-1}} \cdots C_1)$ where all U_i for $i = 1, \ldots, k$ have at most n elements. Let U' be a set of n' elements where $n' > n$. Then $C_{U'} \notin dc(C_{U_k} C_{U_{k-1}} \cdots C_1)$, so it is not realized by a sequence of k n-casts. □

Theorem 24. *In the usual version of CSP with '2-casts', $C_U(A)$ with $|U| > 2$ cannot be achieved for any A which was not common knowledge to begin with.*

5. Examples

Every level of knowledge can be generated in an appropriate system. Practically there may be two kinds of reasons we want to obtain some particular level of knowledge.

(1) We want to have *enough* knowledge in the system so we specify which strings L we want to have included in a realized level $L(A, H)$.

(2) We want to prevent certain processes from knowing some facts about the others. In such a situation we would specify a set L' of the strings we don't want to include in $L(A, H)$. Let us consider now a pair (L, L') as a knowledge specification for our system, and consider how we might realize it.

1. If $L' \cap dc(L)$ is not empty them (L, L') cannot be realized for any formula.

For example if $L = \{K_1 K_2 K_3\}$, $L' = \{K_1, K_3\}$ then there is no system which realizes (L, L').

So a necessary condition for realizability of (L, L') is:

$$\forall m \in L \; \forall m' \in L' \; \neg (m' \leqslant m).$$

Now we have two possibilities:

2. If $dc(\Sigma^s - L') = dc(L)$ then (L, L') uniquely specifies the level, which can be realized for an appropriate formula in an appropriate system.

Let us look at an example. We have some fact F, and we want to create a history H, such that $H \vDash K_A K_B F$, $H \vDash K_B K_C F$, $H \vDash K_C K_A F$ (these three facts already imply that $H \vDash K_A F$, $H \vDash K_B F$, $H \vDash K_C F$, but we also want $H \vDash \neg (K_A K_C F)$, $H \vDash \neg (K_B K_A F)$, and $H \vDash \neg (K_C K_B F)$. So our specification is (L, L'), where $L = \{K_A K_B, K_B K_C, K_C K_A\}$, $L' = \{K_A K_C, K_B K_A, K_C K_B\}$. Notice that in this case, since we exclude the possibility of common knowledge between any pair of processes, our level of knowledge must be finite. In fact these requirements characterize the level of knowledge completely. The level we are looking for is exactly the set $L(A,H) = \{K_A K_B, K_B K_C, K_C K_A, K_A, K_B, K_C, \varepsilon\}$.

We can attain (L, L') by a protocol in which (for example) all A, B, C independently learn about F, A sends a message F to C, C sends F to B, B sends F to A, and all messages are sent asynchronously. In a synchronous system we can achieve (L, L') by a protocol in which first we use a broadcast to bring about common knowledge of the fact $(p \wedge q \wedge r) \rightarrow F$. p is a private fact of A, q is a private fact of B, r is a private fact of C. A broadcasts p to $\{A, B\}$ and later B broadcasts $C_{\{A,B\}} p$ to $\{B, C\}$ creating $N(K_A K_C)$. Similarly first B broadcasts q to $\{B, C\}$, later C broadcasts $C_{\{B,C\}} q$ to $\{A, C\}$ creating $N(K_B K_A)$, finally C broadcasts r to $\{A, C\}$, and then A broadcasts $C_{\{A,C\}} r$ to $\{A, B\}$, creating $N(K_C K_B)$.

3. If (L, L') can be realized but there is $x \in \Sigma^s$ such that $x \notin dc(L)$ and $\neg (\exists m' \in L', m' \leqslant x)$ then the level is not uniquely specified (we can include x in (L, L') but we don't have to).

The smallest level realizing (L, L') is $dc(L)$ and the largest one is $N(L')$ (or $dc(\Sigma^s - L')$).

Let for example $L = \{C_{\{1,2\}} K_3\}$, $L' = \{K_3 K_2\}$. In order to realize (L, L') we can take $dc(L)$. This can be done by taking formula A to be P_3, and a history:

$$s(3, 1, K_3 P_3); r(1, 3, K_3 P_3); bc(1, \{1, 2\}, K_3 P_3).$$

Another possibility is to realize $N(K_3 K_2)$.

We thank Konstantinos Georgatos for a very careful reading of the manuscipt.

References

Aumann R 1976 Agreeing to disagree. *Ann. Stat.* 4: 1236–1239

Chandy M, Misra J 1986 How processes learn. *Distrib. Comput.* 1(1): 40–52

de Jongh D H J, Parikh R 1977 Well partial orderings and hierarchies. *Proc. K. Ned. Akad. Wet.* A80: 195–207

Halpern J, Moses Y 1990 Knowledge and common knowledge in a distributed environment. *J. Assoc. Comput. Mach.* 37: 549–578

Halpern J, Zuck L 1987 A little knowledge goes a long way: simple knowledge-based derivations and correctness proofs for a family of protocols. *Proc. 6th ACM Symp. on Principles of Distributed Computing* (New York: ACM Press) pp. 269–280

Higman G 1952 Ordering by divisibility in abstract algebras. *Proc. Lon. Math. Soc.* 2: 326–336

Lamport L 1978 Time, clocks and the ordering of events in a distributed system. *Commun. ACM* 21: 558–565

Lewis D 1969 *Convention, a philosophical study* (Harvard: University Press)

Moses Y, Tuttle M 1988 Programming simultaneous actions using common knowledge. *Algorithmica* 3: 121–169

Parikh R 1986 Levels of knowledge in distributed computing. *IEEE Symposium on Logic in Computer Science* (New York: IEEE Press) pp. 322–331

Parikh R, Krasucki P 1986 Levels of knowledge in distributed computing, Brooklyn College Dept. of CIS. Technical Report

Parikh R, Krasucki P 1990 Communication, consensus and knowledge. *J. Econ. Theory* 52: 178–189

Parikh R, Ramanujam R 1985 Distributed processes and the logic of knowledge. *Logics of programs. Lecture Notes in Computer Science. Vol. 193* (Berlin: Springer-Verlag) pp. 256–268

Schiffer S 1972 *Meaning* (Oxford: University Press)

Sādhanā, Vol. 17, Part 1, March 1992, pp. 193–220. © Printed in India.

Scalable concurrent computing

NALINI VENKATASUBRAMANIAN*, SHAKUNTALA MIRIYALA† and GUL AGHA

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
*Present address: Hewlett Packard Company, 19111 Pruneridge Avenue MS44UT, Cupertino, CA 95014, USA
†Present address: Vista Technologies, 1100 Woodfield Rd Suite 108, Schaumburg, IL 60173, USA

Abstract. This paper focusses on the challenge of building and programming scalable concurrent computers. The paper describes the inadequacy of current models of computing for programming massively parallel computers and discusses three universal models of concurrent computing – developed respectively by programming, architecture and algorithm perspectives. These models provide a powerful representation for parallel computing and are shown to be quite close. Issues in building systems architectures which efficiently represent and utilize parallel hardware resources are then discussed. Finally, we argue that by using a flexible universal programming model, an environment supporting heterogeneous programming languages can be developed.

Keywords. Scalable concurrent computing; massively parallel computers; systems architectures; heterogeneous programming languages.

1. Introduction

Computers have penetrated into many branches of learning, industry and art. The increasing scope of application domains for computers has brought new demands for computer technology. At the same time, computers and programming methodologies have undergone dramatic changes in the past couple of decades, both in the conceptual view of a program as well as the physical layout of the machine. An analysis of these changes suggests that computer systems may be divided into five generations as shown in table 1 (Hwang & Briggs 1984). The foundational basis of the first four generations of computers is the stored-program concept or the *von Neumann* model. This paper describes fifth generation computer systems and focusses on developments in software technology necessary to realize it.

The outline of this paper is as follows. Section 2 describes the von Neumann model. This model has fundamental limitations for scalability which must be overcome. By scalability, we mean the ability of the system to display an increase in performance

Table 1. Classification of computer systems into generations.

Generation	Hardware advances	Software advances
First	Electromechanical devices	Machine language
Second	Transistors	Assembly, Fortran, Algol
Third	Small and medium scale integrated circuits	Multiprogramming
Fourth	Large scale integrated circuits	Advanced compilers
Fifth	Very large scale integrated circuits	Parallel programming

with the addition of computational resources. We then review concurrent computer architectures and their scalability characteristics. Section 3 reviews universal models for concurrent computing. Specifically, we describe 'actors', a scalable model of concurrent programming which overcomes the limitations of the von Neumann model. A number of problems, such as composability of independent systems, debugging and effective system management etc., arise in concurrent systems. Section 4 addresses problems in realizing concurrent computation, namely, new techniques for managing the computational resources of large numbers of concurrent processors. Section 5 discusses the use of actors to provide interoperability in programming environments – i.e., to provide a software technology which permits use of multiple programming paradigms. The final section discusses some ongoing research projects and future directions in scalable concurrent computing.

2. Concurrent computer architectures

In the von Neumann model, the hardware consists of a *processor*, a *memory module* (also called *storage*) and a *link* to communicate between them as shown in figure 1. A *program* is a sequence of instructions stored in memory and executed by the processor. In order to execute an instruction at least three communications have to pass through the link – the *instruction* (control), *operands* (data from memory to processor) and *results* (stored back in memory). However, having a single communication bus poses bandwidth limitations on the amount of information that can be transmitted between the memory and processor; this imposes a performance bottleneck, traditionally known as the *von Neumann* bottleneck.

The hardware components in a von Neumann computer operate by performing a

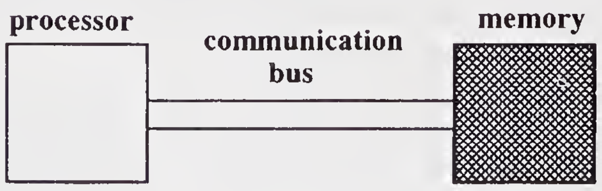


Figure 1. The von Neumann model.

sequence of transitions on a *state*[1]. The sequence of transitions is specified by the user through a program as a function of the state. Note that the programmer specifies not only the transitions which must be performed but also a *strict order* on the execution of these transitions by means of program instructions. In the model presented above, a single processor is the sole unit of computational power which results in the following two limitations:

Physical limits on how close components can get. Technological advances have led to the miniaturization of components as well as increase in processing speed. In fact, processor speeds have doubled every eight years.

Economic considerations on the cost of a single, highly sophisticated processor. One solution is to have a number of inexpensive units, all working simultaneously so that the overall throughput of the machine is increased. This is called *parallel processing.*

Industrial automation accompanied by the expanding size and nature of problems which have to be handled by computers has placed a premium on speed, efficiency and accuracy of computation. Weather prediction, nuclear physics, modelling physical and geological phenomena, genetic code mapping, space exploration, flight and space vehicle control, expert systems for medical diagnosis are representative problems. Applications run on computers may be numeric or non-numeric. In general, the use of computers is expanding from just number crunching to include *symbolic processing*, processing of non-numeric data such as pictures, text and sentences, and open information systems which are characterized by continuous information flow between autonomous units.

Nondeterminism is a key feature of the real world systems directly modelled in an open system. Nondeterminism has many sources. For example, when searching a very large database for some specific information, it may be necessary to explore many possible choices. It is not clear when we will arrive at a solution or which paths to investigate in the process. Searching many paths for a possible solution places a premium on performance. To satisfy this growing demand for performance we can exploit parallelism.

Computational models are inspired both by concern for representing real world problems and by architectural considerations. In building architectures which exploit parallelism, there is an obvious tradeoff between the number of processing units and their complexity. Variations in the degree of coupling of these two factors has given rise to a number of computational models for enhancing performance. At one end of the spectrum, there are machines with large numbers of very simple processors. The degree of concurrency is very high, but the power and complexity of each individual processing unit is low. The Connection Machine (Hillis 1985) is one such computer available in the market. On the other hand, there are multiprocessor architectures which use sophisticated processing units. The Encore Multimax is an example of a commercial multiprocessor machine.

2.1 *High performance uniprocessor architectures*

Supercomputers exploit parallelism by using multifunction pipelines; i.e., their central processing units (CPU) consist of multiple functional units. Multiple functional units are processing units that can perform more than one function (operation) simultaneously. Each functional unit is equipped with a set of registers to store the

[1] A state is the data stored in the memory of the computer at a given point in time.

data resulting from computation on that functional unit. The multiple functional units also share a set of general purpose registers that communicate with main memory. Results from one functional unit can be used as input to other functional units via a high speed internal bus. This brings us to the concept of *pipelined computation*. In pipelined parallelism, a complex operation **O** is broken up into a sequence of simpler and faster operations $o_1, o_2, \ldots, o_n$. Different operations can execute on different data units simultaneously; for instance, the output from o_1 can be automatically piped into the functional unit exeeuting o_2, while new data is input to o_1. A number of existing computers exploit pipelined parallelism in their processing units, for example, CRAY, CDC-7600 and IBM 360/91.

Studies on supercomputer development over the past years have indicated that there is only a marginal increase in the sequential speed of supercomputers. The earliest Cray machines operated at 160 megaflops (floating point operations per second), while the more recent Cray X-MP operates at a peak performance of 210 megaflops. Current supercomputers have multiple processors (up to 4) in addition to multifunctional units, but do not provide an order of magnitude change in performance.

The *workstation* industry has benefited heavily by the introduction of RISC (Reduced Instruction Set Computer) processors which simplify the design of processors by transferring some of the work to software. For example, the original VAX uniprocessor exhibits a performance of the order of 1 MIP (million instructions per second) for typical uniprocessor RISC-based workstations. Today, this figure has increased to 20–25 MIPS. Typically, RISC processors also exploit pipelined parallelism.

2.2 *Parallel architectures*

Parallel architectures are classified based on the manner in which they exploit concurrency into *data parallel machines* and *control parallel machines*. *Data parallelism* allows the same operations to be independently performed on each element of a large aggregate of data. By contrast, *control parallelism* allows multiple threads of execution. Control parallelism is more general: implicit in control parallelism is the fact that each thread of execution may involve distinct data.

2.2a *Data parallel models*: Machines based on the data parallel model are frequently known as SIMD (single instruction multiple data) machines (see figure 2).

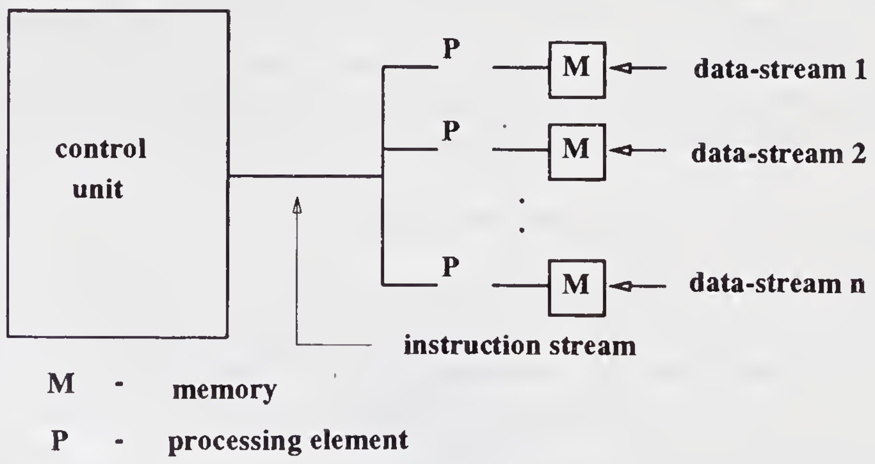
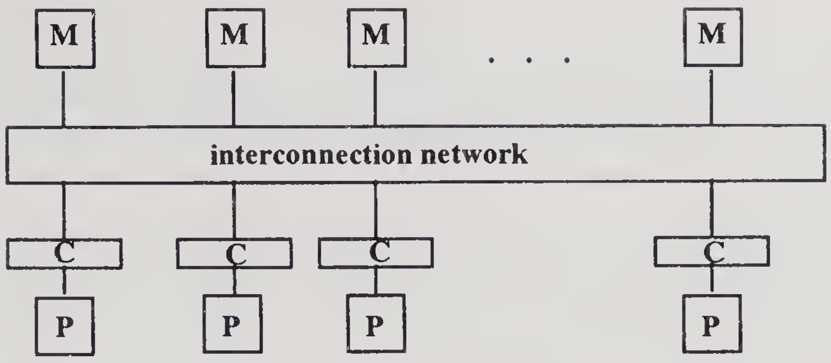


Figure 2. The data parallel model.

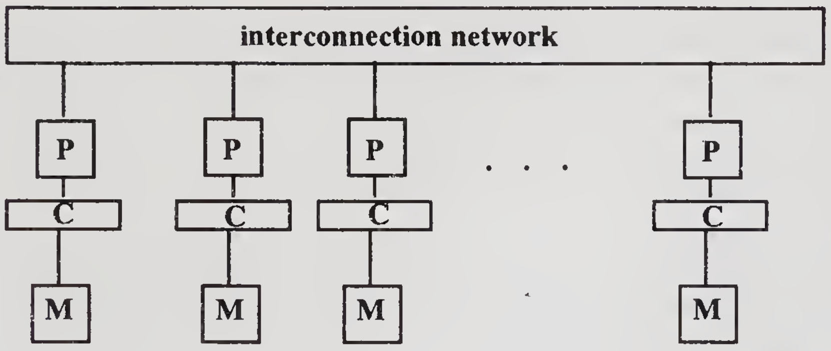


M	-	**memory**
P	-	**processing element**
C	-	**local cache**

Figure 3. The shared memory model.

Each instruction stream operates on multiple units of data such as a vector or array instead of on a single operand. A single control unit coordinates the operation of the multiple processors. Synchronous computers using global clocks are quite special purpose and rather restrictive in their model of concurrent computation. This approach is exemplified by the connection machine (Hillis 1985) and conventional array processors.

2.2b *Control parallel models*: Control parallel computers can be divided into two broad classes: *shared memory* machines and *message-passing* concurrent computers (also called *multicomputers*) (figures 3 and 4). Shared memory computers have multiple processors and share a global memory. For efficiency reasons, each processor also has a local cache.[2] Separate caches create the problem of maintaining consistency



M	-	**memory**
P	-	**processing element**
C	-	**local cache**

Figure 4. The message passing model.

[2] A cache is a fast, expensive memory unit between the processor and the main memory and is used to hold frequently accessed data.

between caches when processors may modify shared data in their local caches simultaneously. The shared memory computers which have been built typically consist of 16 to 32 processors. Large numbers of processors create increased contention for access to the global memory. The contention for shared information increases as the computing resources increase. As a result, shared memory architectures are not scalable (Dally 1986).

Multicomputers have evolved out of work done by Charles Seitz and his group at Caltech (Athas & Seitz 1988). Configurations of multicomputers with only 64 computers exhibit performance comparable to conventional supercomputers. Multicomputers use a large number of small programmable computers (processors with their own memory) which are connected by a message-passing network.

Depending on the amount of memory per component, multicomputers may be divided into two classes, namely, *medium-grained multicomputers* and *fine-grained multicomputers*. Two generations of medium-grained multicomputers have been built. A typical first generation machine (also called the cube or the hypercube because of its communication network topology) consisted of 64 nodes and delivered 64 MIPS. Its communication latency was in the order of milliseconds.[3] On the other hand, a typical second generation medium-grained multicomputer has 256 nodes and is projected to carry out 2·5 K MIPS and has a message latency in the order of tens of microseconds. Third generation machines are currently being built and are expected to increase the overall computational power by two orders of magnitude and reduce message latency to a fraction of a microsecond (Athas & Seitz 1988).

The frontiers of multicomputer research are occupied by work on fine-grained multicomputers. Two projects building experimental fine-grained multicomputers are the J-Machine project by William Dally's group at MIT (Dally & Wills 1989, pp. 19–33) and the Mosaic project by Charles Seitz's group at Caltech (Athas & Seitz 1988). A number of other innovative architectures such as dataflow, reduction machines and logic programming engines have been proposed (Arvind & Culler 1986; Shapiro 1987).

Concurrent computing involves *partitioning* and *communication* of tasks. Partitioning involves the splitting up of a process into many threads of computation which may execute in cooperation. Threads belonging to the same process interact with each other and the underlying system must provide for communication between them. It is also necessary that the threads executing in parallel synchronize or cooperate with each other without conflicts. Thus at a higher level *coordination* is required.

Partitioning may be *explicit* where the user divides the tasks into a few large threads or heavyweight objects which may execute in parallel. This is referred to as *coarse grained partitioning*. In this approach, however, concurrency inherent in each of the tasks is not fully exploited. The need for a high degree of parallelism leads to *fine grained partitioning*. Typically, partitioning a task into the grained threads is performed by the compiler. Here, the task is split into many small threads or lightweight objects (fine grained parallelism). Very large scale integration (VLSI) is an elegant medium for expressing this form of parallel computation. The computational

[3]Communication latency is a measure of the time delay involved in transmitted information from source to destination. As communication latency increases, more time is required to transmit information from one processing unit to another. As a result, the performance of the system drops.

quanta obtained by partitioning a task may need to communicate with each other and this communication introduces additional overhead. The presence of smaller threads introduces more communication which may offset the benefits of parallelism.

Communication is needed to transfer information from one processor to another. Communication strategies may be synchronous or asynchronous.

- *Synchronous communication*: Both the sender and receiver must be available before communication begins. A real life example of synchronous communication is a telephone that waits for users at both ends to be available at the same time before conversation can occur.
- *Asynchronous communication*: Here, the sender and receiver can operate independently, the sender can send information without waiting for the receiver to be ready to accept the information. Communication that takes place through a postal system is a real-world example of asynchronous communication. Any communication between two people or two points goes through some post office. Note that it is not necessary to have a centralized controller in such a system. The presence of buffers at the receiver allows us to store the messages arriving from possibly different senders. This kind of communication is called *buffered asynchronous communication.*

When there are two or more processes executing in parallel, data dependencies dictate the threads which need to cooperate or *synchronize.* This cooperation or information transfer is achieved by means of efficient synchronization primitives implemented in hardware.

With increased parallelism, we believe that the current wave of technologies and mechanisms will cause a shift in programming paradigms. In particular, from a programming language standpoint, we are observing a shift from textual to visual (graphic) programs (Miriyala 1991). From an implementation standpoint, there is a shift from static to dynamic resource management (Venkatasubramanian 1991).

3. Universal models for scalable concurrent systems

3.1 *Inadequacy of existing software strategies*

Computers available in the market today are not sufficiently powerful to compete with the pressing demand for computational speed and power. Hardware advances have been so rapid that today it is possible to think about packing massive computational power into a few inches of silicon. The advent of VLSI technologies has also brought forth highly concurrent hardware. This alone is not sufficient to satisfy our requirements. The computational capacities of these hardware modules can be exploited only through efficient software methods to program these machines. The challenge, therefore, lies in the fact that software technologies have not scaled up in proportion to potential hardware advances. Radical software techniques have to be developed to concurrently program thousands, and in the future, millions of processors to work in concert.

Most familiar programming languages are based on the von Neumann model of computation. A major drawback of von Neumann languages (languages based on the von Neumann model of computation) is that the architectural model of the machine is the basis of the programming model of the language. Thus the programmer is subjected to a serious constraint in forcing his/her programs to reflect the underlying architecture when one should be worrying about the most effective way to specify the problem without imposing any artificial constraints. Only then can he/she fully harness the computational power of the underlying hardware.

Another handicap of traditional languages is their difficulty in combining existing program modules in a number of ways to achieve different functionalities.[4] Furthermore, programs written in these languages are designed for deterministic computations and traditional programming strategies break down when we have to deal with nondeterminism.

Scalability, in the context of parallel computation, implies that given a program with sufficient parallelism, it will be possible to increase the performance of the system by adding physical resources. In a scalable system, no alteration of the application program is necessary to exploit the benefit of the added resources. Scalability is a fundamental requirement of *open systems*, i.e. of information systems which permit continuous influx of input from new and different sources.

Realistic modelling of natural phenomena and real world systems as open systems is possible because an open system has the following attributes:

(1) decentralization;
(2) inherent concurrency;
(3) organizational cooperation.

One can view a parallel computation model as a set of abstractions that capture the semantics and functionality of concurrent program execution. This problem has been approached from different perspectives and various models of concurrency have been proposed by language theorists, complexity analysis and architects of parallel machines. We analyse some of these models with a view towards determining their suitability as a general purpose parallel programming model for scalable systems.

There are a number of closely related approaches to developing universal models of concurrency. A conservative language strategy to defining a model of concurrency is to start with sequential processes and add communication primitives. One language-based model that does this is Communicating Sequential Processes (CSP) developed by Tony Hoare and others (Hoare 1978). In CSP, communication is *synchronous*, i.e., a source process cannot send a communication until a destination process is ready to accept it, and the process topology is *statically* determined, i.e., the process configuration cannot be altered during program execution. The significant limitations of such a model include the need to specify all concurrency explicitly, the need to predetermine resources to be used, and the need to fix synchronization points.

We focus on the three universal models – a programming model, a complexity model, and an architectural model. A universal model facilitates the development of various applications without any concern for the underlying architectural configuration. If a universal model can be used, it insulates hardware and software developments from each other: it is possible to use the advances in hardware (software) without having to be overly concerned about equivalent advances in software (hardware). Interestingly, the proposed universal models have similar underlying features. We will discuss the following classes of universal models:

[4]This property is called *composability*.

A programming model: *Actors* is a model of concurrent object oriented programming with active agents developed primarily by programming language theorists.
A complexity model: The *bulk synchronous parallel* (BSP) model proposed by Valiant (1990, pp. 103–111) is an intermediate model to bridge the gap between concurrent programming languages and parallel architectures. It deals with complexity issues involved in *efficient universality* and is a useful model for designers of parallel algorithms.
An architectural model: The *parallel machine interface* (PMI) developed by Bill Dally and his group at MIT (Dally & Wills 1989, pp. 19–33) is an implementation oriented machine model. The PMI aims at achieving efficient hardware implementations of primitive abstractions.

It is possible to measure the cost of implementing different operations in various programming models on the basis of the primitive operations on each of these universal models. We discuss them in greater detail below.

3.2 *The actor model*

The *actor model*, first proposed by Hewitt (1977) and later developed by Agha (1986) captures the essence of concurrent computation in distributed systems at an abstract level. In the actor paradigm the universe contains computational agents called *actors*, which are distributed in time and space. Each actor has a conceptual location (its *mail address*) and a *behavior* as illustrated in figure 5.

The only way one actor can influence the actions of another actor is a send the latter a communication. An actor can send another actor (or itself) a communication only if it knows the mail address of the recipient. On receiving a communication, an actor processes the message and as a result may cause one or more of the following events:

- creation of a new actor,
- change its behavior, and,
- send a message to an existing actor.

In asynchronous communication, the sender and receiver need not coordinate message delivery. If two messages sent to an actor arrive at their destination simultaneously, there must be some mechanism to serialize the incoming communications and execute both messages. To ensure this, every actor is equipped with a mailbox that queues any communications received. Therefore, communication in the actor model is *asynchronous* and *buffered*. The size of the buffers is theoretically unbounded.

Although actors is inherently an asynchronous model, it is possible to simulate synchronous models as specializations of the actor model. A important characteristic of communication in the actor model is the ability to communicate mail addresses. Thus the interconnection topology of the system is capable of changing continuously. This adds to the reconfigurability and flexibility of the actor model; for example, it allows resource management decisions such as object to processor mapping to be directly programmed. Another property of the actor model is the *guarantee of delivery*. i.e., messages in the system will eventually reach their destination actor. This implies that mail in transit cannot be indefinitely buffered.

Actor execution is graphically expressed in terms of event-diagrams (see figure 6). An event-diagram is a pictorial representation of the arrival order of events within a thread of execution and the causal relationship between different threads of

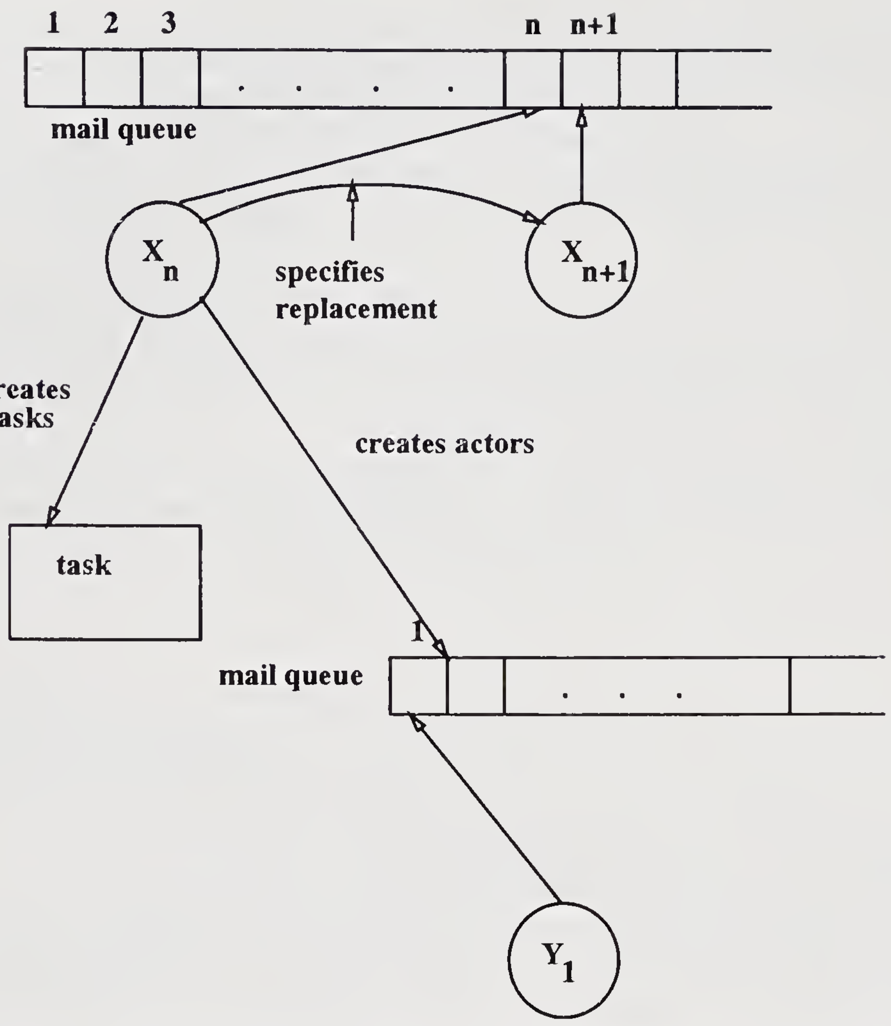


Figure 5. An abstract representation of actor transitions: When an actor processes the *n*th communication, it determines the replacement behavior which will process the $(n + 1)$th communication. The mail address of the actor remains unchanged. The actor may also send communications to specific target actors and create new actors.

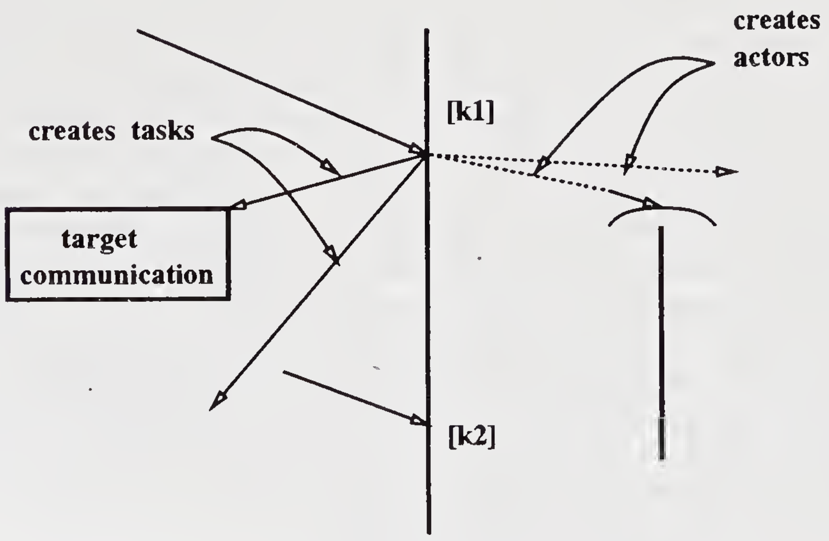


Figure 6. Each vertical line represents the linear arrival order of communications sent to an actor. In response to processing the communications, new actors are created and different actors may be sent communications which arrive at their target after an arbitrary delay.

computation. The thick vertical line in figure 6 represents the actor on a linear temporal scale with time floating from the top of the line to the bottom. Events that occurred earlier lie above those that occur later. Event diagrams bring out the concept of local time and local state in the actor model. Causality connections form the fundamental synchronization mechanism in the actor model.

The actor model is well-suited for fine grained computation because of its dynamicity and its ability to create new actors inexpensively. It may sometimes to necessary to sequentialize task execution using one of the following methods:

(1) *Introduce causality constraints* between different threads of execution. This does not reduce the grainsize of a task, but causes sequentiality that allows controlled resource allocation.
(2) *Execute a given thread of control sequentially* instead of in a functional fashion. Either user-defined or compiler-derived annotations could be used to specify the sequentiality. For example, we could have both sequential and parallel versions of a recursive method. The parallel version creates new tasks for every recursive call. The sequential version is called when the argument to the recursive call falls below a certain value. The sequential method definition executes sequentially to completion without creating new tasks.

Baude & Vidal-Naquet (1991, pp. 184–195) have shown that the asynchronous, message passing actor model is as powerful as the traditional PRAM or Parallel Random Access Memory model (used to analyse the complexity of parallel algorithms) which is a synchronous, shared memory model.

In the PRAM model, program execution is deterministic and can utilize an unbounded number of logical processing units. The logical processing units communicate via a global shared memory. Efficient parallel programs are defined as those programs that demonstrate an exponential speedup with a polynomial increase in the number of processing units. Such *efficiently parallelizable programs* are classified under the *NC* class of programs.

Efficiently actor parallelizable problems or NC_{actor} are defined as those problems for which there exists an actor program whose time complexity (the length of the execution chain created by the actor) is a polylogarithmic function of the input size and whose size complexity (a measure of the space needed to process the message) is a polynomial function of the input size. Baude & Vidal-Naquet (1991, pp. 184–195) provide a simulation of actors by PRAM and vice-versa. In particular, they show:

$$NC_{\text{PRAM}} \subseteq NC_{actor}$$

In summary, the actor model is a convenient perspective for the programmer of the parallel machine due to its flexibility and simplicity (programming with abstractions). However, it does not model some architectural characteristics – such as the overheads involved in message passing.

3.3 *The bulk synchronous parallel model*

Valiant (1990, pp. 103–111) proposed the bulk synchronous parallel (BSP) model as a suitable bridge between parallel languages and architectures. The BSP model consists of three units:

- components – which are computing or memory units;
- router –responsible for point to point message passing;

- periodic synchronization – that performs synchronization of all or a subset of the processors with a periodicity L.

To achieve *efficient universality* results, we must be able to model the performance of the system as a relation. If h is the number of messages sent or received in a computation on a given processor and g is the sum of all computations in the system per second divided by the number of data words delivered per second, it is assumed that all communications are delivered in time gh. Let the startup latency or cost be s. The router sends and is sent at most h messages in a superstep. Such supersteps are also called *h-relations*. Therefore, the cost to realize an h-relation is $gh + s$. This technique of modelling the performance of a system as a relation offers parameterized controllability and is one of the basic features of the BSP model that makes it amenable to complexity analysis.

Memory components in the BSP model are distributed across the processing components and therefore access to every computational unit is equally likely. To allow efficient symbolic to physical address mapping, the BSP model adopts a pseudo-random mapping or hashing mechanism. The assumption is that known hash functions are used and they can be computed locally and inexpensively.

Synchronization in the BSP model is carried out periodically using *bulk synchronization*. The process of bulk synchronization, from which the model derives its name, is as follows. Initially, the mean delay to execute each operation is roughly estimated. After the estimated time has elapsed, an elected processor sends out a sync detect signal to all the other processes along a spanning tree. If synchronization is achieved, we can proceed with the next phase of computation. Otherwise, a new timeslot is allocated and the process is repeated till synchronization is achieved.

The tasks generated are executed in a sequential fashion but all the tasks generated can execute in parallel. The tasks involved in a bulk synchronization must wait until the synchronization has been achieved for further continuation of the executing task. However, it is possible for processing components to switch this mechanism off and execute without waiting for synchronization. As a consequence, task granularity in the BSP model is controllable by varying the periodicity of synchronization, L. As L increases, the granularity of the program also increases. The type of algorithms most naturally cxpressed using the BSP model are PRAM programs.

The BSP is a possible model for the designer of a parallel algorithm. Valiant (1990, pp. 103–111) demonstrates how the BSP model can be embedded on theoretical models like PRAM as well as architectural models like networks of systems. However, the level of abstraction in BSP makes it difficult to specify optimizations that may be necessary for efficiency – in particular, no specific language interface or architectural issues can be addressed. Furthermore, the model limits communication costs in an architecture by assuming the feasibility of efficient bulk synchronization. Thus BSP would need to be modified before it could be used as a model of a realistic scalable system.

3.4 *The parallel machine interface*

In Dally & Wills (1989, pp. 19–33), a universal machine model has been proposed to support various parallel models of computation, the *Parallel Machine Interface* (PMI). Any machine model, sequential or parallel, must have abstractions representing hardware components, i.e., memory, instructions and instruction sequencing. One such general-purpose abstraction in a sequential machine that has been very successful is the notion of stack-based storage allocation. Complex modelling primitives lack flexibility and generality, and are less amenable to optimizations and therefore, it is important to keep the modelling primitives simple and straightforward.

What are the kinds of abstractions needed to support a parallel model of computation? Three basic requirements of any parallel model are *communication, synchronization* and *naming*. Most proposed models of parallel computation like dataflow (Arvind & Culler 1986), static message passing, shared memory, parallel logic programming (Goto *et al* 1988, pp. 208–229) and concurrent object oriented programming improvise on similar implementations of these primitive abstractions. The goal is therefore to design efficient hardware implementations of these primitive mechanisms that are portable across different programming systems.

A parallel machine interface isolates the issues of programming models from the details of machine organization and implementation. This abstract model of parallel systems is then embellished with special features in order to support a particular programming paradigm efficiently on a specific target architecture.

A parallel architecture interface, called Pi has been proposed based on the PMI (Wills 1990) and the implementation of several machine models has been illustrated. An abstract machine architecture has also been proposed for the same and we will use this abstract architecture implied by Pi in the discussion below.

The PMI views storage as a collection of data units or *segments* which are logically related. It makes no assumptions about how segment names are interpreted in an implementation. However, some translation mechanism must be built to support segment addressing and access. The cost of accessing a segment is directly dependent on the translation scheme chosen.

The PMI currently assumes a message passing model of communication, but mechanisms like communication via shared memory or RPC (remote procedure calls) can also be implemented. The communication network does not take a stand on routing and buffering mechanisms. As in the actor model, the PMI assumes that all messages are eventually delivered (albeit with an arbitrary delay due to network latency) and that there is no message order preservation.

The PMI models the three primitive operations in a parallel system as follows:

(1) *Communication* is represented by means of a message send. Other forms of communication like shared memory reads and writes are more complex mechanisms implemented in terms of message sends.
(2) *Synchronization* is of two kinds: *data synchronization*, which is needed when there is a data dependence between executing tasks, and *control synchronization*, which requires that a task must complete before another can begin execution. The mechanism used to implement safety and correctness in synchronization is *actor firing* and progress after synchronization is guaranteed by *I-structure accesses* (Arvind *et al* 1987).
(3) *Naming* issues in most models can be implemented as some form of translation from a logical name to a physical address in the memory of the system. The model assumes that the translation is embedded within the message injection and reception mechanisms.

In addition, the Pi model defines primitives to provide information regarding the relative proximity of objects to a particular node.

The abstract machine implied by the Pi model consists of a set of nodes interconnected by means of a communication network. Although the Pi model is fine-grained (where grain size is measured in terms of the node size), it should be observed that fine-grained systems are *upward-compatible* with larger grained systems.

Table 2. Universal models of parallel computation.

	BSP	Actor	PMI
Storage model	Memory and components	Actors as Abstract Data Type (ADT)	Logical collection of data units called *segments*
Communication	Asynchronous point to point	Asynchronous point to point	Asynchronous point to point
Synchronization	Bulk synchronization	Causality and history-sensitivity	Access attributes
Naming	Pseudo-random mapping of symbolic names to physical addresses	Unique actor names	Nametable for address translation
Process/object Topology	Unclear	Dynamic creation Dynamic configuration	Dynamic creation Dynamic configuration
Expressing Parallelism	Sequential tasks Specified	Parallelism default Implicit synchronization	Fine-grained threads Explicit synchronization
Locality	Hash function distributes memory components	Actors are a unit of locality	Primitives to support spatial locality

In other words, architectures for the fine-grained computation can support larger grain size efficiently (but not necessarily vice-versa). Using the Pi model, it is possible to measure the cost of implementing different operations in various programming models in terms of the primitive operations on the PMI. The translation between PMI and actors is one-to-one. It is worth noting that costs measured in terms of the PMI model apparently closely correspond to actual machine costs.

3.5 *A comparison of universal models*

Table 2 gives a summary of the three different universal models we have considered. As is evident from all the models, some attributes of parallel computation are mandatory to any model irrespective of whether it has an analytical or architectural flavour, i.e., memory, communication and synchronization. However, the degree of specification with respect to other attributes depends largely on the level of abstraction that the model is trying to achieve.

3.6 *A hierarchical view of parallel computation*

Good scalability characteristics are achieved only if the problem size scales up with the number of processors. In other words, merely increasing the number of processors keeping the problem size constant is expensive. Furthermore, scalability and complexity analysis on these models must also consider the implications of the algorithm and its mapping on the underlying architecture (see, for example, Singh *et al* 1991). A hierarchical view of a parallel machine is shown in figure 7.

The purpose of the hierarchical model is to provide layers of abstraction in a parallel system. When dealing with high level issues like algorithm design or programming language issues, we would like to abstract away from implementation

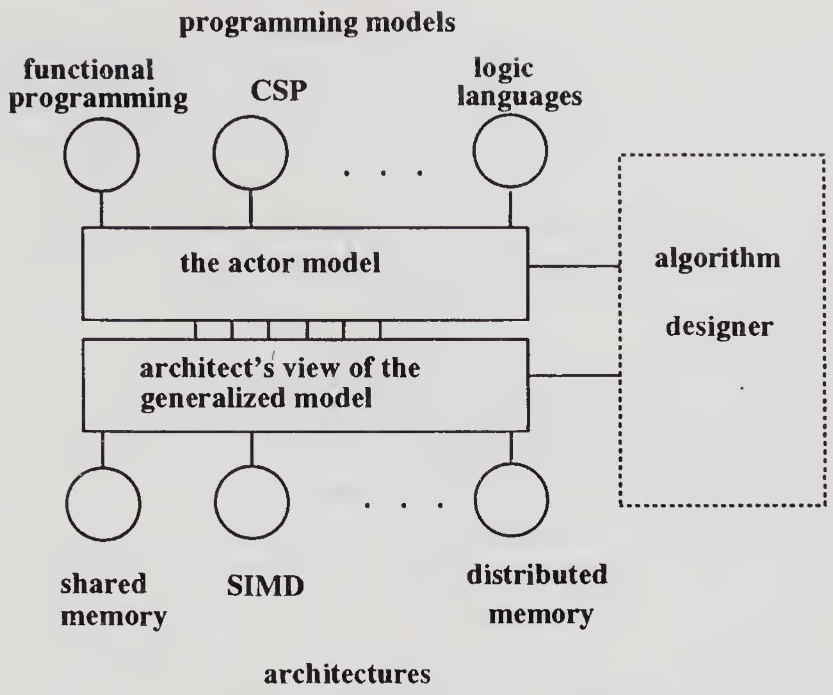


Figure 7. A hierarchical view of parallel systems: Existing models do not consider the implications of algorithm design in the scalability and complexity analysis.

or architectural details like the communication subsystem or resource management. However, in order to discuss resource management issues without referring to any particular programming language or architecture, we must isolate features specific to parallelism from issues specific to the framework in which parallelism is being exploited. The hierarchical view in figure 7 illustrates the different levels at which a parallel system can be viewed for different purposes (Venkatasubramanian 1991).

We now consider parallel resource management with the actor model as the programmer's view of the world and the parallel machine interface as the underlying model of a highly concurrent machine.

4. Resource management for scalable systems

In this section, we focus on the actual implementation of scalable parallel programs on a highly parallel machine. In particular we elaborate on techniques for the effective utilization of hardware.

Resources refer to the hardware and software components of a computer system which are required in order to solve a specific problem. *Resource management* involves techniques and mechanisms used to efficiently allocate, utilize and coordinate these resources. Parallelism creates new complexities for resource management, for example, the communication network is one of the most critical of execution resources. In fact, a bottleneck is scaling concurrent computers is not limitations in the computational power of individual processors, but the costs and delays associated with transferring information from one processor to another. Thus resource management strategies must try to reduce the communication traffic.

Another source of complexity in parallel systems is that different rcsources may be needed simultaneously and these resources must all be available. Furthermore, in a large scale concurrent system, dynamic allocation of resources is necessary because computations may need to be automatically divided into large numbers of sub-computations. In actor systems, such division happens each time concurrent sub-requests are made as a result of processing a given request (Agha & Hewitt 1987). With every fork in the computation, execution resources must be provided to the sub-computations created. Often the behavior of sub-computations cannot be determined in advance. Different strategies for subdividing resources lead to different results. An improper allocation may lead to a condition where a useful subcomputation cannot proceed due to the non-availability of resources, called the *dangling sub-computation problem.*

Dynamically spawning off tasks may give rise to expanding resource requirements limited only by their physical availability. By serializing executions, resources can be exclusively allocated to the executing process and there are no resource contentions. However, little parallelism is exploited by this scheduling strategy. The other extreme aims at extracting all the parallelism inherent in the application by using a suitable programming paradigm. In such cases, studies have determined that many programs are "embarassingly parallel" (Sargeant 1986). Excessive parallelism leads to inordinate resource utilization and may have an even more serious outcome – deadlock due to unavailability of resources for any of the parallel tasks to continue execution. There is an obvious crossover point between the degree of parallelism exploited and effective resource utilization in a concurrent system. The goal, then, is to be able to set up a flexible *throttle* which will allow us to increase or reduce the degree of parallelism at will (Arvind & Culler 1986; Sargeant 1986).

Another aspect of resource management is controlling the use of resources at the application level. Some problems exhibit an exponential growth in the number of possible paths which could lead to a solution; it is infeasible to explore all of them. This in turn means that differing amounts of resources must be allocated to different paths. The assessment of how fruitful a particular path is changes over time as one assesses the intermediate results along the path. This is one reason resources need to be controlled and re-allocated dynamically.

Other complications in a real implementation arise from the presence of prioritized execution of processes, conflict resolution, deadlock handling and synchronization. Effective resource management in parallel machines is a combination of:

Static analysis: The text of the program is analysed before execution at compile-time to detect infomation such as useless concurrency via dependence analysis, rough estimates of resource requirements, data placement and partitioning.
Heuristics: It is often sufficient to approach the resource management problem conservatively. In fact, mechanisms implemented to guarantee completeness or totality may degrade machine performance, thereby defeating the purpose of an efficient resource management strategy. Heuristics designed to achieve effective resource management concentrate on detecting sections of program execution critical to performance and target their restructuring efforts at those sections where the payoffs would be significantly large.
Reconfiguration strategies: The system must be capable of dynamically detecting "congestion points" and handling them, when appropriate, by alteration of the existing system configuration.

Resource management issues can be divided into actor/process management, memory management and I/O (input–output) management.

4.1 *Actor management*

Actor placement and migration is governed by mechanisms implemented to handle locality and load balancing.

4.1a *Locality*: A process in one processing unit may need to communicate with a process or object on another processing unit. For example, in figure 8, processes P1 and P3 on different processing units may need to communicate with each other. Similarly, processes P2 and P4 also interact with each other. The allocation of processes as illustrated in figure 8 reduces processor utilization caused by tasks waiting for the result of a communication. Frequent communication between processes on different processors also increases network contentions. Minimizing network traffic is important even in the presence of high speed networks, where the network bandwidth is the major limiting constraint on communication. Locality directed scheduling policies, as in figure 9, reduce interprocessor communication, thereby decreasing network traffic.

Memory locality is a property of the pattern of a program's references to memory. References which are tightly grouped together in terms of addresses are said to have spatial locality. References close together in time are said to have temporal locality. Three distinct classes of locality can be identified in user programs:

(1) temporary objects that are usually intermediates during computation;
(2) fairly long lived data structures whose lifetime is a significant portion of the program's lifetime;
(3) permanent structures such as the runtime system's routines and structurcs.

The above hierarchy of relative object lifetimes is well portrayed and cxploited in the generational garbage collection schemes discussed later.

4.1b *Scheduling*: The scheduling problem in multiprocessor machines is that of distributing multiple threads of control to execute on the available processing resources. Threads which share a lot of data and threads which communicate frequently might yield better throughput if scheduled to execute on the same processor. But this means that the execution of these tasks is serialized, inhibiting possible parallelism.

A straightforward policy is to statically schedule tasks to execute on specific processors despite the fact that better choices could be made dynamically after the execution scenario is more well defined. An alternative is to perform dynamic

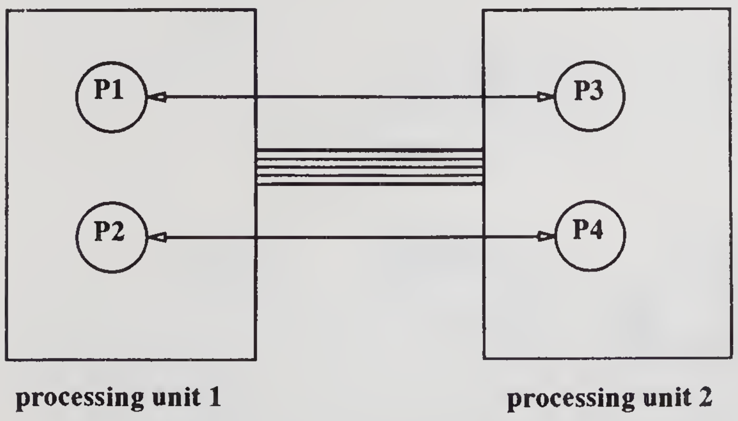


Figure 8. A system that does not exhibit locality. Communication which occurs across processors blocks the network which is a critical resource.

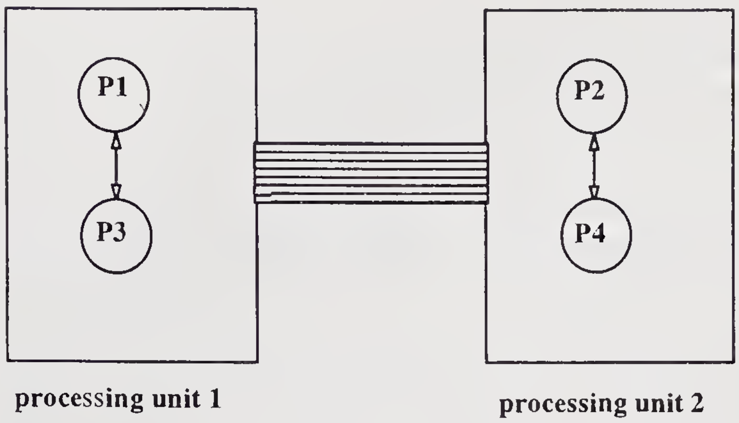


Figure 9. A system that exploits locality. Locality based scheduling strategies avoid unnecessary communication overhead.

scheduling by having a global pool of tasks which need to be executed, from which processors can pick their next to execute. However, a centralized job queue may disrupt the locality inherent in the application.

4.1c *Load balancing*: Load balancing is the task of keeping the processors of a parallel machine uniformly busy. Figures 10 and 11 illustrate this concept.

For effective load balancing, each processor needs to know the degree of load in every other processor, or at least, some controller needs to maintain load information in order to make load balancing decisions. In the latter case, there is likely to be a contention for this centralized resource (i.e. the controller). If there is a static interconnection scheme between processors in a system, each processor need only maintain nearest neighbor information.

Locality and load-balancing are contradictory constraints that need to be weighed against each other for determining an optimal tradeoff between dispersion and aggregation (Athas 1987). Locality aims at placing related objects in close physical proximity, thereby mitigating the frequent communication costs. Load balancing efforts are geared toward splitting and distributing work and do not encourage "clubbing" of computational nodes or threads. In order to make effective use of resources, we must exploit the right amount of locality needed to mask the communication latency. The degree of object mobility required to achieve this balance and mechanisms to deal with this have been explored in Jul (1989).

4.1d *Network activity*: An important concern of network behavior in a parallel system is that the links of the network should be kept uniformly busy with data

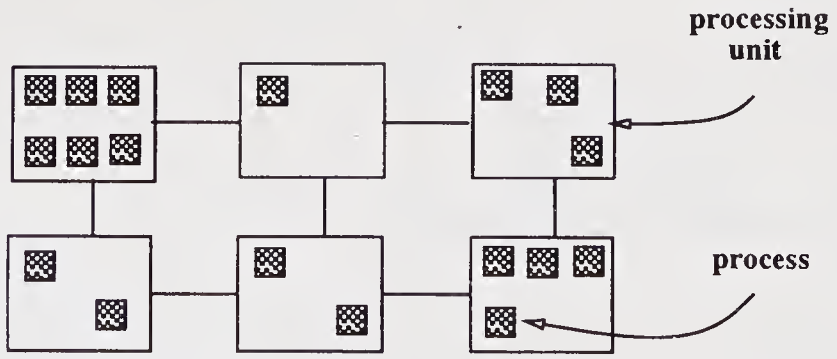


Figure 10. Processors before load balancing. Note that the throughput of the system is limited by its slowest component. Processes within a processing unit execute sequentially and the heavily loaded processor becomes a bottleneck.

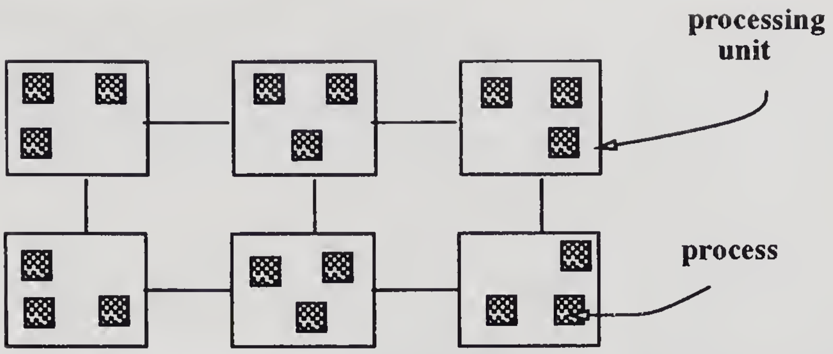


Figure 11. Processors after load balancing. The uniform distribution of computational units improves the system throughput.

objects. Non-uniform distribution of objects may result in bottlenecks at frequently accessed nodes, thereby delaying other communications. Often a process will create a stream of information used by another process. Such processes are called *producer* and *consumer* processes respectively. An important factor for proper utilization of the network is that message dispatch and reception rates of a producer and its consumer of the communication be compatible. Consider the scenario where we have a producer–consumer relationship between objects. In order to exploit concurrency, we assign them to different nodes. If these nodes are adjacent to each other in a mesh, we have a single link between them and this may result in non-uniform traffic in the network. To avoid this problem, we can place the two objects on nodes which are far apart. However, note that message reception at the consumer is serialized and placing the nodes far apart in the network would block the network which is a shared resource in the system. Thus there is a contradictory argument that presses for locality. If the producing and consuming rates are well-matched, we could exploit concurrency without excessive traffic on the network.

4.1e *A distributed namespace*: We assume that communication in a distributed memory system can be modelled as a series of message sends and that messages are routed by means of a fast, efficient routing network to their destinations. Local communication is accomplished through primitive messages[5]. We advocate having a uniform address space across all nodes in a network. All entities in the system are referred to by a virtual name that is uniform over the entire system, giving us a uniform namespace over the entire system. A uniform namespace brings us additional flexibility in resource handling: it permits us to migrate objects to support actor placement, migration and garbage collection. The cost of this flexibility is the overhead associated with name translation (virtual name to physical address). Each access to the object must query the translation table and it is therefore vital that all translations be completed with a relatively low latency. Obviously, we should avoid virtual name to physical address translation for objects which reside locally and as far as possible for remote objects through intelligent compilation techniques and optimizations.

4.2 *Memory management*

Storage management is used to allocate space for an object when it is created and subsequently reuse this allocation space when it is no longer needed. Memory which is not accessible is referred to as *garbage* and the reclamation of garbage is *garbage*

[5] Primitive messages are messages that a processor sends to itself.

collection. An efficient storage management scheme plays a crucial role in enhancing the efficiency of a programming system. The two major issues involved are storage allocation and reclamation. Automatic storage management schemes require that the runtime system be capable of recognizing a shortage of memory and reclaiming unused memory for reallocation. The actor-based storage model adopted here assumes the following operating systems support services:

(1) a mechanism that can allocate and deallocate contiguous blocks of memory;
(2) abstractions needed to access objects in memory with a single virtual address across a distributed global object namespace;
(3) effective support for relocation of objects either within the heap or across the network (object migration).

4.2a *Garbage collection*: There have been a number of schemes and algorithms for performing garbage collection (Ungar 1984, pp. 157–167; Baker & Hewitt 1987). The following steps are either implicitly or explicitly performed in all the algorithms.

(1) Identification of accessible objects;
(2) reclamation of the inaccessible memory objects;
(3) compaction of memory to improve locality.

Some of the traditional garbage collection algorithms take time proportional to the size of memory which makes them inefficient as the size of memory increases. The additional time taken to detect and reclaim all the unreachable memory cells may also destroy the interactive response of the system. Storage reclamation involves both temporal and spatial overhead. In addition, stringent time restrictions imposed by interactive programs necessitates efficient memory management mechanisms.

General requirements of a garbage collection algorithm:

(1) good interactive response;
(2) capability to reclaim circular structures;
(3) requires minimal hardware support;
(4) meshes well with virtual memory;
(5) causes only a small impact on execution overhead.

With the help of operating system services, a number of low-level memory management details can be abstracted thus enabling us to design a generalized garbage collection algorithm. We will now briefly describe the traditional GC mechanisms and extend them to parallel machines.

In *Reference Counting* schemes, every cell (object) in memory is associated with a field called the *reference count* which is a count of the number of references to the cell. The reference count of a cell is continuously updated as pointers to the cell are created or destroyed. When the reference count of a cell becomes 0, the object is no longer referenced and can be collected. Reference counting mechanisms are incremental in nature. Therefore, computation is not interrupted for a significant period of time. A major flaw, however, is their inability to handle circular garbage (because the reference count of any cell in a circular structure can never be zero). Also, when an object is collected, additional work is done in decrementing the reference counts of any child cells. This process can be potentially unbounded. There have been attempts to modify reference counting algorithms to accommodate various implementation issues (Deutsch & Bobrow 1976; Bevan 1987, pp. 273–288; Watson & Watson 1987, pp. 432–443; Goldberg 1989, pp. 313–320; Ichisugi & Yonezawa 1990).

The *Mark and Sweep* algorithm detects garbage by halting the mutator (application program) and then performing two (optionally three) phases. A predetermined root set is used to *mark* all the reachable objects in the first phase, the second phase reclaims dead objects one at a time by walking down the entire memory space and a third, optional phase compacts memory to avoid fragmentation. This algorithm collects all available garbage and uses only one extra bit per word for reclamation purposes. However, the running time of mark and sweep is proportional to the size of memory, and hence this method is impractical for large memory sizes.

The *Stop-and-Copy* garbage collector divides the available memory in a system into two regions – the *from-space* and the *to-space*. Only one of these regions, i.e., the from-space, is available to the mutator at any time. The copying algorithm (Fenichel & Yochelson 1969; Cheney 1970) traverses all the reachable records in the from-space starting at a root pointer set and copies these records into a separate area of memory, the to-space, that is currently unused by the mutator. Forwarding pointers are placed in the from-space to redirect any references to the swapped record. At the end of collection, the only records in to-space are the reachable ones. The from- and to-space pointers are swapped and the mutator now works on the new from-space. Note that the amount of work performed is proportional to the number of reachable cells whereas the traditional mark-and-sweep algorithms take time proportional to the size of memory. The performance of the copying algorithm is unaffected as the size of memory increases.

Generational Garbage Collection (Liebermann & Hewitt 1983) is an extension of the copying garbage collector that takes advantage of two significant observations:

(1) high rate of infant mortality – A young object is more likely to die than an object that has survived for a while;
(2) newer records are more likely to point to older records than vice-versa. An older record points to a newer record only if it is altered (reassigned) after it is initialized. This flavour of reassignment is rather unlikely in many languages. Generational garbage collection is claimed to be highly suitable for such languages.

Records with a similar age are grouped together in a contiguous area of memory. Once we have segregated objects on the basis of their age, reclamation efforts can be concentrated on the youngest generation. The drawback of this scheme is what is known in garbage collection literature as the *Tenuring Problem*. Old objects that die may take a long time to be reclaimed. Even worse, they may never be reclaimed until the occurrence of an offline collection. If objects are promoted too quickly, there are more objects present in higher generations. Collecting higher generations results in significant pauses and clever promotion policies are necessary to mask this unavoidable tradeoff.

4.2b *Multiprocessor* GC *algorithms*: The desire to exploit concurrency among tasks in a job complicates storage management. This is due to the need for synchronization between the various tasks in order to ensure their correctness and consistency. Programming environments for highly parallel machines, it appears, are largely centred around dynamic resource management schemes which require sophisticated memory allocation and reclamation schemes. Furthermore, one can think of storage reclamation itself as a concurrent set of processes.

All this concurrency requires coordination between the parallel subtasks of a single job which in turn requires efficient and reliable communication. Deficiencies of traditional algorithms in a parallel setting have been detailed upon in Hudak & Keller (1982, pp. 168–178). Many of the insufficiencies arise from the fact that the traditional algorithms halt the computation process, preventing realtime response by the system. In other words, the collector (thread of control that is responsible for garbage collection) is initiated after the mutator(s) (threads of control corresponding to the application program) have been halted.

The main overhead in parallel GC occurs in maintaining intraprocessor and interprocessor references to an object. If this is not done, it is possible for an object to be collected even while a reference to it exists, violating the correctness criteria of the algorithm.

• *GC in shared memory machines*

One of the earliest parallel mark-and-sweep algorithms on a shared memory has been discussed in Steele (1975). A dedicated storage reclamation processor handles garbage collection of a memory space that it shares with a mutator. In the general case, this algorithm could be extended to handle any number of mutators that share a single address space. Hence, this algorithm is applicable to a tightly coupled system. The correctness proof for a similar mark-and-sweep based algorithm was developed in Dijkstra *et al* (1978).

Locking mechanisms must be implemented to prevent two processors from simultaneously manipulating the same memory location. Extensions of the stop-and-copy algorithm and generational GC implemented on shared memory machines are presented (Appel *et al* 1988, pp. 11–20; Pallas & Ungar 1988, pp. 268–277).

• *GC in distributed memory machines*

In distributed architectures, each processing unit is associated with its own local memory. A global communication network connects the different processing units together. As intra-processor communication is orders of magnitude faster than interprocessor communication, we are forced to exploit locality and manipulate data in the local memory in a distributed memory machine.

There are two levels of garbage collection on distributed memory machines, i.e., local and global garbage collection. Global garbage collection is a systemwide GC involving all processors in the system. Local garbage collection is carried out locally on each processor without inhibiting the processing activities on another processor. There are costs associated with both local and global GC. The interval of a global garbage collection is dictated by the first processing unit that needs memory. Other processors have to cooperate even if they have sufficient local memory to continue work. The overhead of synchronization – orchestrating a global start and stop could be substantial, especially when the collection process has to be real time. In other words, the deviations in the time taken by different processing elements to run out of space must be more well-balanced. Hardware techniques like logic and signal lines or software mechanisms like barrier synchronization can be used to reduce the synchronization overhead. Local garbage collection requires a reference management table to determine whether an object is wholly local or not. The entries in this table should be periodically updated and eventually reclaimed.

The authors' group is currently studying algorithms for distributed memory management and a hierarchical approach to memory management has been presented in Venkatasubramanian (1991).

4.3 *I/O management*

While advances in processor architecture and memory/register management strategies have enhanced performance in parallel machines, commensurate improvements in I/O performance have not been made. The disparity between the performance of the I/O subsystem and the other subsystems in current parallel architectures must be reduced. This problem is acute in the data parallel model explained earlier, where the physical dimensions of the machine pose a limitation on the amount of information which can enter or leave the surfaces of the machine.

In a three-dimensional physical world with n processors along each dimension, a cube has a total of n^3 processors with 6 faces. As all the input/output to a system configured as a cube occurs through the face of the cube, the n^3 processors communicate with the outside world through the $6n^2$ processors on the face of the cube. It must be noted that this is the hypothetical best case for a cube configured system. This yields an I/O limitation of $6n^{2/3}$ for loading information from external sources to the processors in a cube simultaneously from all the $6n^2$ entry points into the cube. To illustrate the above discussion, consider a cube with a million processors. Thus an order of 17 basic time units would be needed to load one unit of information into each of the processors in a million processor multicomputer with optimal I/O.

It is possible to avoid the I/O bottleneck for specific applications (Kung 1986, pp. 49–54), but the bottomline remains that we must focus efforts on improving the I/O subsystem performance to the level that it does not affect throughput. Real-time visualization and animation are examples of I/O limited applications.

Interoperability in programming environments, i.e., the ability to switch from one programming framework to another, will play an important role in future distributed systems. Multiparadigm programming environments are software technologies that will permit a smooth transition among programming paradigms.

5. Multiparadigm programming environments

In order to address the problems of concurrent computation, a number of important programming paradigms have been developed. Our research focuses on providing a flexible basis for providing efficient execution of important programming constructs inspired by different linguistic paradigms in a single framework.

5.1 *Declarative programming*

In *declarative programming*, there is no notion of instruction sequencing – we "declare" what is to be computed rather than how this computation must be done. Separating logic from control is an important step toward achieving abstraction and modularity. All control or computational issues governing machine behavior are relegated to the language implementation. Two approaches to declarative programming are functional and logic programming.

Declarative programming is a more radical approach to concurrent computing. Unlike the CSP model where the concurrency and state transformations are explicit, declarative models are implicitly parallel. An important feature of declarative languages is that variables in these languages correspond to values; they have no notion of a computational history or state. The absence of state resolves a number of determinacy issues, for example, there are no cache consistency problems. However, the ability to create shared, modifiable data structures cannot be cleanly integrated into declarative models. State must be perpctually passed through procedural abstractions such as functions or relations. This is inadcquate for programming in the large system which requires support for data abstraction.

Functional and concurrent logic programming suggest important linguistic constructs and programming techniques. Important languagc features in these paradigms, such as higher-order functions and pattern-matching, can bc defined in terms of actor primitives (Agha 1989, pp. 1–19) and used as nceded.

5.2 *Concurrent object oriented programming*

Object Oriented Programming (OOP) is another programming paradigm whose roots go as far back as Simula (a simulation language), with significant contributions made by developments in languages such as Smalltalk and Flavors (Wegner 1990). Modern object oriented programming has been influenced by a number of individuals and concepts which makes it difficult to give a universally acceptable definition of object oriented programming. A simplified definition is given in Madsen (1987):

A program execution is regarded as a physical model, simulating a behavior of either a real or imaginary part of the world.

Here, the universe is viewed as being composed of many passive entities with functions which transform them. Structuring mechanisms in OOP include classification and specialization. Classification enables us to group diverse objects based on some common characteristics. Specialization supports the specification of objects as modifications of existing specifications. Abstraction and inheritance are language mechanisms which allow this structuring. Object-oriented programming provides encapsulation: the user sees a simplified interface consisting of a set of operations permissible on a structure.

5.2a *Abstraction*: Abstraction is a powerful tool which allows programmers to manage complexity by dealing with high level concepts before dealing with their representation. Object oriented languages are designed to support both data and procedural abstraction. *Procedural abstraction*, common to all modern programming languages, allows different actions to be grouped into a single name. *Data abstraction* is a data structuring and packaging mechanism in which small subsystems which have certain common characteristics are grouped together to form a larger subsystem.

In other words, an object is a unit of encapsulation which hides from the user, implementation details (i.e., the representation) of the data structure as well as the operations performed on it. The user need only be concerned about what operations may be performed on the object. An advantage of object-oriented programming is the easy maintenance, modifiability and portability of the code. Actors are similar to objects in that they encapsulate a local state and a set of operations. However, actors differ from objects in several ways. Perhaps the most important distinction is that actors are inherently concurrent – many actors may be active at the same time. By contrast, in sequential object oriented languages, only one object may be active at a time.

5.2b *Inheritance*: Inheritance is a mechanism which facilates code sharing and reusability. Codc is structured in terms of superclasses and subclasses. Inheritance mechanisms allow a class to borrow the functionality (methods) of its superclass and specialize it. Code sharing can lead to name conficts – the same identifier may be bound to procedures in an object and in its class. Object oriented languages differ in how such name conflicts are handled. One proposal, advanced by Jagannathan and Agha (see, for example, Agha & Jagannathan 1991), is to allow programmers to define different possible inheritance mechanisms in a single linguistic framework by providing the ability to define and manipulate the underlying name bindings.

The power of Concurrent Object Oriented Programming (COOP) comes from the ability to develop abstractions which hide the details and complexities of concurrency. COOP systems are larger systems of communicating actors and they can be built in terms of actor primitives. The inherent locality in distributed systems is modelled naturally through a modular design. A judicious choice of abstractions and primitives which express concurrency in a manner transparent to the user is of crucial importance. It has been observed that the main contribution of OOP to concurrency is to provide reusable abstractions which can hide the low level details of partitioning, synchronization and communication from the user (Lim & Johnson 1989). Thus, COOP systems illustrate the power of the actors as building blocks. Another approach to COOP can be found in ABCL (An Actor Based Concurrent Language) (Yonezawa 1990).

5.3 *Reflection*

In the normal course of execution of a program, a number of objects are implicit. In particular, the interpreter or compiler being used to evaluate the code for an object, the text of the code, the environment in which the bindings of identifiers in an object are evaluated, and the communication network are all implicit. As one moves from a higher level language to its implementation language, a number of objects are given concrete representations and can be explicity manipulated at the lower implementation level (for a more detailed discussion see Agha 1990, 1991, pp. 1–59).

The dilemma is that if a very low level language is used, the advantages of abstraction provided in a high-level notation are lost. Alternately, the flexibility of a low-level language may be lost in a high-level language. A *reflective architecture* addresses this problem by allowing us to program in a high-level language without losing the possibility of representing and manipulating the objects that are normally implicit (Maes 1987). *Reification* operators can be used to represent at the level of the application, objects which are in the underlying architecture. These objects can then be manipulated like any other objects at the "higher" application level. *Reflective* operators may then be used to install the modified objects into the underlying architecture. Reflection thus provides a causal connection between the operations performed on this representation and the corresponding objects in the underlying architecture.

In a COOP system, the evaluator of an object is called its meta-object. Reflective architectures in COOP have been used to implement a number of interesting applications. For example, Watanabe and Yonezawa (see Yonezawa 1991) have used it to separate the logic of an algorithm from its scheduling for the purposes of a simulation: in order to build a virtual time simulation, messages are time-stamped by the meta-object and sent to the meta-object of the target which uses the time-stamp to schedule the processing of a message or to decide if a rollback to a previous state is required. Thus the code for an individual object need only contain the logic of the simulation, not the mechanisms used to carry out the simulation; the specification of the mechanisms is separated into the meta-objects.

In summary, reflection provides a way of integrating various language paradigms. By using the underlying representations, one can define programming constructs and their interrelation as first-class objects. Thus a multiparadigm programming environment may be created for heterogeneous computing. Components of a system may use language constructs which are more suited to the computations they are carrying out.

6. Discussion

There are a number of ongoing efforts to achieve the fifth generation computing objective. The US High Performance Computing and Communication (HPCC) Initiative (1991–1996), funded at a level of over 4 billion dollars, attempts to address fundamental problems in high performance computing. Major emphasis is placed on the development of hardware and software technologies required for scalable, parallel computing systems with a performance of trillions of operations per second on a wide range of important applications. The European ESPRIT project is placing emphasis on models of computing based on the human brain and parallel architectures.

Similar efforts are being conducted in Japan in the New Information Processing Technology funded by the Japanese Ministry of International Trade and Industry. Areas of research include establishing sound theoretical foundations for flexible information processing technologies using distributed and cooperating agents or actors, development of massively parallel computing technologies (a million processor architecture) and new application domains for such technologies.

The US HPCC program has identified a large number of problems critical to science and engineering, the so-called *grand challenge* problems. Addressing these grand challenges will require orders of magnitude increase in computational power. For example, in biomedical investigations, research on genes causing cancer requires management of data of the order of billions of molecular units. The fastest supercomputers available in the market today will require hundreds of years of processing time to yield the necessary information. Similarly, fundamental research in physics and chemistry, and superconductor research require accurate simulations to predict the transformations of materials under varying conditions. Weather prediction and information about severe changes in atmospheric conditions are other computationally intensive problems.

Scalable concurrent computing is a fundamental enabling technology required to deliver the promise of high performance computing. We are just beginning to address some of the problems involved in using massively parallel processing.

The authors would like to acknowledge Chris Houck, Subrata Mitra and Rajendra Panwar for their comments on the paper. The work described in this paper has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Development Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90–07195).

References

Agha G 1986 *Actors: A model of concurrent computation in distributed systems* (Cambridge, MA: MIT Press)

Agha G 1989 Supporting multiparadigm programming on actor architectures. In *Proceedings of Parallel Architectures and Languages Europe, Vol. II: Parallel Languages (PARLE'89) Lecture Notes in Computer Science. Vol. 366* (Berlin: Springer-Verlag) pp. 1–19

Agha G 1990 Concurrent object oriented programming. *Commun. ACM* 33 (9): 125–141

Agha G 1991 The structure and semantics of actor languages. In *Foundation of object-oriented languages. Lecture Notes in Computer Science. Vol. 489* (Berlin: Springer-Verlag) pp. 1–59

Agha G, Hewitt C 1987 Actors: A conceptual foundation for concurrent object-oriented programming. In *Research directions in object oriented programming* (Cambridge, MA: MIT Press)

Agha G, Jagannathan S 1991 Reflection in concurrent systems: A model of concurrent continuations, Technical Report, University of Illinois at Urbana Champaign

Appel A, Ellis J, Li K 1988 Real-time concurrent collection on stock multiprocessors. In: *SIGPLAN'88 Conference on Programming Language Design and Implementation* Atlanta, GA

Arvind, Culler D E 1986 Dataflow architectures. In *Annu. Rev. Comput. Sci.* 1: 225–253

Arvind, Nikhil R, Pingali K 1987 I-structurcs: Data structures for parallel computing, Technical Report Computation Structures Group Memo 269, Massachusetts Institute of Technology, Cambridge, MA

Athas W 1987 *Fine grain concurrent computations*, Ph D dissertation, Computer Sciencc Department, California Institute of Technology (also published as technical report 5242:TR:87)

Athas W, Seitz C 1988 Multicomputers: Message-passing concurrent computers. *IEEE Comput.* pp. 9–23

Baker H, Hewitt C 1977 The incremental garbage collection of processes, Technical Report Memo AL-454, Massachusctts Institute of Technology, MIT Artificial Intelligence Laboratory

Baude F, Vidal-Naquet G 1991 Actors as a parallel programming model. In *Proceedings of the 8th Symp. on Theoretical Aspects of Computer Sciences: Lecture Notes in Computer Science. Vol. 480* (Berlin: Springer-Verlag) pp. 184–195

Bevan D I 1987 Distributed garbage collection using reference counting. In *Parallel architecture and languages Europe: Lecture Notes in Computer Science. Vol. 259* (Berlin: Springer-Verlag) pp. 273–288

Cheney C J 1970 A nonrecursivelist compacting algorithm. *Commun. ACM* 13: 677–678

Dally W J 1986 *A VLSI architecture for concurrent data structures* (Kluwer: Academic Press)

Dally W J, Wills S 1989 Universal mechanisms for concurrency. In *Parallel architecture and language Europe: Lecture Notes in Computer Science. Vol. 365* (Eindhoven: Springer-Verlag) pp. 19–33

Deutsch P, Bobrow D G 1976 An efficient, incremental, automatic garbage collector. *Commun. ACM* 19: 522–526

Dijkstra E W, Lamport L, Martin A J, Scholten C S, Steffens E F M 1978 On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM* 21: 966–975

Fenichel R R, Yochelson J C 1969 A lisp garbage collector for virtual memory systems. *Commun. ACM* 12: 611–612

Goldberg B 1989 Generational reference counting: A reduced-communication distributed storage reclamation scheme (1989). In *SIGPLAN'89 Conference on Programming Language Design and Implementation* (Portland, OR: ACM Press)

Goto A, Sato M, Nakajima K, Taki K, Matsumoto A 1988 Overview of the parallel inference machine architecture. In *Fifth generation computing systems* (Tokyo: ICOT)

Hewitt C 1977 Viewing control structures as patterns of passing messages. *J. Artif. Intell.* 8: 323–364

Hillis D 1985 *The connection machine* (Cambridge, MA: MIT Press)

Hoare C A R 1978 Communicating sequential processes. *Commun. ACM* 21: 666–677

Hudak P, Keller R M 1982 Garbage collection and task deletion in distributed applicative processing systems. In *ACM Symposium on LISP and Functional Programming* (Portland, OR: ACM Press)

Hwang K, Briggs F 1984 *Computer architecture and parallel processing* (New York: McGraw Hill)

Ichisugi Y, Yonezawa A 1990 Distributed garbage collection using group reference counting, Technical report, University of Tokyo, Dept. of Information Science

Jul E 1989 *Object mobility in a distributed object-oriented system*, Ph D thesis, University of Washington

Kung H T 1986 Memory requirements for balanced computer architectures. In *Proc. of the 13th Annual Symposium on Computer Architecture* (New York: IEEE Press)

Lieberman H, Hewitt C 1983 A real-time garbage collector based on the lifetimes of objects *Commun. ACM* 26: 419–429

Lim J, Johnson R 1989 The hcart of object oriented concurrent programming. *Sigplan Notices* 24(4): 165–167

Madsen O L 1987 *Block-structure and object oriented languages* (Cambridge, MA: MIT Press)

Maes P 1987 *Computational reflection*, Ph D thesis, Vrije University, Brussels, Belgium (Technical Report 87–2)

Miriyala S 1991 *Visual representation of actors using predicate transition nets*, Master's thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, Urbana, IL

Pallas J, Ungar D 1988 Multiprocessor smalltalk: A case study of a multiprocessor-based programming environment. In *SIGPLAN Conference on Programming Language Design and Implementation*

Sargeant J 1986 Load balancing, locality and parallelism control in fine-grained parallel machines, Technical Report UMCS-86-11-5, Dept. of Computer Science, University of Manchester

Shapiro E 1987 *Concurrent prolog: Collected papers, Series in Logic Programming* (Cambridge, MA: MIT Press)

Singh V, Kumar V, Agha G, Tomlinson C 1991 Scalability of parallel sorting on mesh multicomputers. In *Proceedings of the International Parallel Processing Symposium*

Steele G L 1975 Multiprocessing compactifying garbage collection. *Commun. ACM* 18: 495–508

Ungar D M 1984 Generation scavenging – a non-disruptive high performance storage reclamation algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA

Valiant G 1990 A bridging model for parallel computation. *Comput. ACM* 33: 103–111

Venkatasubramanian N 1991 *Hierarchical memory management for parallel machines*, Masters's thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, Urbana, Il (forthcoming)

Watson P, Watson I 1987 An efficient garbage collection scheme for parallel computer architectures. In *Parallel Architectures and Languages Europe, Lecture Notes in Computer Science. Vol. 259* (Berlin: Springer-Verlag) pp. 432–443

Wegner P 1990 Concepts and paradigms of object-oriented programming. In *OOPS Messenger* 1(1): 7–87

Wills D S 1990 *Pi: A parallel architecture interface for multi-model execution*, Ph D thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts

Yonezawa A 1990 *ABCL: An object-oriented concurrent system* (Cambridge, MA: MIT Press)

Author Index

Subject Index