DTIC FILE COPY (2)

UNCLASSIFIED MASTER COPY FOR REPRODUCTION PURPOSES

SECURITY CLASSIFICATION OF THIS PAGE

AD-A218 836

DTIC ELECTE FEB 23 1990 S B D

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARO 26779.1-EL-AI
6a. NAME OF PERFORMING ORGANIZATION Center of Excellence in AI University of Pennsylvania — 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U. S. Army Research Office
6c. ADDRESS (City, State, and ZIP Code) Dept. of Computer & Information Science 200 S. 33rd Street Philadelphia, PA 19104-6389	7b. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U. S. Army Research Office — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAL03-89-C-0031
8c. ADDRESS (City, State, and ZIP Code) P. O. Box 12211 Research Triangle Park, NC 27709-2211	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
Communicating Shared Resources: A Model for Distributed Real-Time Systems (MS-CIS-89-26)

12. PERSONAL AUTHOR(S)
Richard Gerber and Insup Lee

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim technical	FROM ____ TO ____	May 1989	30

16. SUPPLEMENTARY NOTATION The view, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Real-time systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The timing behavior of a real-time system depends not only on delays due to process synchronization, but also on resource requirements and scheduling. However, most real-time models have abstracted out resource-specific details, and thus assume operating environments such as maximum parallelism or pure interleaving. This paper presents a real-time formalism called communicating Shared Resources (CSR). CSR consists of a programming language that allows the explicit expression of timing constraints and resources, and a computation model that resolves resource contention based on event priority. We provide a full denotational semantics for the programming language, grounded in our resource-based computation model. To illustrate CSR, we present a distributed robot system consisting of a robot arm and a sensor.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) — 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

COMMUNICATING SHARED RESOURCES: A MODEL FOR DISTRIBUTED REAL-TIME SYSTEMS

Richard Gerber and Insup Lee

MS-CIS-89-26
GRASP LAB-178

Department of Computer and Information Science
School of Engineering and Applied Science
Philadelphia, PA 19104-6389

COMMUNICATING SHARED RESOURCES: A MODEL FOR DISTRIBUTED REAL-TIME SYSTEMS

Richard Gerber
and Insup Lee

MS-CIS-89-26
GRASP LAB-178

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

May 1989

[illegible]

[illegible]

Acknowledgements: This research was supported in part by ONR grant N000014-89-J-1131, NSF grants IRI86-10617, DCR 8501482, DMC 8512838, MCS 8219196-CER and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027, DAAL03-89-C-0031.

Communicating Shared Resources: A Model for Distributed Real-Time Systems *

Richard Gerber and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

May 7, 1989

Abstract

The timing behavior of a real-time system depends not only on delays due to process synchronization, but also on resource requirements and scheduling. However, most real-time models have abstracted out resource-specific details, and thus assume operating environments such as maximum parallelism or pure interleaving. This paper presents a real-time formalism called Communicating Shared Resources (CSR). CSR consists of a programming language that allows the explicit expression of timing constraints and resources, and a computation model that resolves resource contention based on event priority. We provide a full denotational semantics for the programming language, grounded in our resource-based computation model. To illustrate CSR, we present a distributed robot system consisting of a robot arm and a sensor.

1 Introduction

During the past several years there has been rapid progress in the development of formal semantics for real-time programming languages. However, most research has treated language as an abstraction, quite isolated from any valid operating environment. Thus, often unrealistic assumptions are made about a formalism's underlying computational model. Such assumptions range from the overtly optimistic (*e.g.*, a one-to-one assignment of processes to processors) to the bleakly pessimistic (*e.g.*, all interleavings of process executions are possible). These assumptions rarely hold in practice, and using them to reason about a real-time system's temporal properties can often lead to incorrect conclusions.

*This research was supported in part by ONR N000014-89-J-1131.

It is now understood that models based on pure interleaving semantics are unsuitable as real-time formalisms. Since interleaving concurrency cannot adequately portray simultaneous actions, it fails to permit reasoning about a distributed system's temporal properties. Several real-time process algebras have addressed this issue, among which are ECP [5], TCSP [16] and Timed Acceptances [19]; each of the semantics underlying these formalisms can successfully portray "true" concurrency. However they each suffer from the same defect, in that they place no resource constraints on their underlying computational models. Further, they each permit n-way event synchronization between processes which, while algebraically pleasing, has an unfortunate consequence: processes may delay indefinitely, waiting for communicating partners that simply do not exist. For example, assume that a system of two processes, P_1 and P_2, exclusively share some event a. Even if they successfully synchronize on a, there still exists a possibility that they may use a to synchronize with an elusive P_3 – although the system consists only of P_1 and P_2. In [16] the hiding operator can force the desired synchronization. Thus one may trade observational information for correct temporal properties, which seems a fairly high price to pay.

Recently there has been increasing interest in the *maximum parallelism* view of concurrency, first advanced in [17]. In [11] it was coupled with a linear-history semantics [3], and used to model many temporal properties of Ada [18]. The maximum parallelism model circumvents the problem of unnecessary idling; if two processes are ready to communicate, the communication cannot be arbitrarily delayed. To accomplish this, event synchronization between processes is limited to one sender, and one receiver. Lately, variants of this semantics have been used to model a real-time version of *occam* [7], Statecharts [8] and a design language for distributed, reactive systems [13].

The main defect in "pure" maximum parallelism is that it assumes unlimited computational resources: To enforce the constraint of "no unnecessary idling," each process must be mapped to its own, dedicated processor. Thus the sharing of resources, with all its attendant scheduling issues, can neither be specified nor analyzed. This is unfortunate, since most real-time systems do, in fact, share their resources among many processes [10].

To address this problem, we have developed a real-time formalism called *Communicating Shared Resources*, or CSR. CSR's underlying computational model is *resource-based*, where a resource may be a processor, an Ethernet link, or any other constituent device in a real-time system. At any point in time, each resource has the capacity to execute only a single action. However, a resource may host a set of many processes, and at every instant, any number of these processes may compete for its availability. "True" concurrency may take place only *between* resources; on a single resource, the actions of multiple processes must be interleaved. To arbitrate between competing events, CSR employs a priority-ordering among them. This priority scheme forms the heart of our semantics.

Quite recently there has been some exploration of priorities in both untimed and real-time formalisms (see [2, 9, 13]). However, these studies have treated a limited subset of the problem – that of the priority-based guarded command, such as *occam*'s PRI ALT [15]. CSR's priority semantics can successfully give meaning to this construct. However, CSR addresses the much more general issue of resource-sharing by multiple processes, processes that may contain many such prioritized guarded commands, and that, over time, may continuously have their priorities altered.

This paper is organized as follows: Section 2 gives an overview of CSR, with both its syntax and its informal semantics. In section 3 we extend the basic CSR formalism by supplying higher-level communication primitives. Section 4 presents an example of a real-time, robotics system, modeled with the extended CSR language. In section 5 we provide the mathematical semantics for our language. Finally, in section 6, we state our future objectives.

2 Overview of CSR

The CSR language provides the foundation for our real-time specification method, and all of our higher-level constructs are derived from it. In several ways it syntactically resembles the variants of real-time CSP found in [7] and [11]. Yet it includes many additional features that take full advantage of our priority semantics. Furthermore, it has the capacity to specify many constructs quite common in real-time systems, such as timeouts, periodic processes, and exception-handling.

2.1 Events

Our basic unit of computation is the *event*, which we use to model both interprocess communication as well as local process execution. All of our events are drawn from the finite event alphabet Σ. Here we use the following conventions:

- The letters a, b and c range over Σ.
- The letters A, B and C range over $\mathcal{P}(\Sigma)$.
- The Greek letters Δ and Γ range over $\mathcal{P}(\mathcal{P}(\Sigma))$.

When executed, each event consumes exactly one time unit. However, this does not imply that all actions require exactly the same amount of time. On the contrary, the event should only be considered a common infinitesimal unit, a building-block with which more complex functions are constructed.

2.1.1 Communication and Computation

In CSR all communication between processes is strictly one-to-one, and performed by synchronizing on common events. So if a represents such a synchronizing event, there is a single process P that may utilize a to denote a "write" action, and a single process Q that may use a to denote a "read." Syntactically, P would contain the "$a!$" statement, while Q would contain the "$a?$" statement. When both processes agree to communicate, they both simultaneously execute the a event. At that point we say the event is *resolved.*

As we have stated, events may also be used to explicitly represent local computation within a process; that is, any action that requires possession of the processor's resources. Syntactically, one unit of a local computation is simply denoted by an event name, such as "a." Semantically it is treated as a communication event that has implicitly been resolved.

2.1.2 Event Priority

Both of the two processes that synchronize on an event have their own priorities associated with it. In other words, each communicating event has two priorities, one for the "reader" and the the other for the "writer." Two functions, $PRI_i \in \Sigma \rightarrow \mathcal{N}$ and $PRI_o \in \Sigma \rightarrow \mathcal{N}$ represent the respective priority mappings. If a process uses event a to model a read action, the priority of that action is $PRI_i(a)$. Similarly, if a process uses a to model a write, the associated priority is $PRI_o(a)$. On the other hand, if the event a is employed to represent a computation unit. the function $PRI\ (a)$ yields the priority of a.

2.2 The Syntax of CSR

The following is a complete grammar for the CSR language:

⟨system⟩	::=	⟨resource⟩ | ⟨system⟩ || ⟨system⟩
⟨resource⟩	::=	{⟨process⟩}
⟨process⟩	::=	⟨stmt⟩ | ⟨process⟩ & ⟨process⟩
⟨stmt⟩	::=	**wait** t | **skip** | $a?$ | $a!$ | **exec**(a, m, n) | ⟨stmt⟩ ; ⟨stmt⟩ | ⟨guard_s⟩ |
		⟨within_s⟩ | ⟨interrupt_s⟩ | ⟨loop_s⟩ | ⟨every_s⟩
⟨guard_s⟩	::=	[⟨guard⟩ → ⟨stmt⟩□ ... □⟨guard⟩ → ⟨stmt⟩△ **wait** t— ⟨stmt⟩] |
		[⟨guard⟩ → ⟨stmt⟩□ ... □ ⟨guard⟩ → ⟨stmt⟩]
⟨guard⟩	::=	a | $a?$ | $a!$
⟨within_s⟩	::=	**within** t **do** ⟨stmt⟩ **when** t → ⟨stmt⟩ **od**
⟨interrupt_s⟩	::=	**interrupt** a **do** ⟨stmt⟩ **when** $a?$ → ⟨stmt⟩ **od**
⟨loop_s⟩	::=	**loop do** ⟨stmt⟩ **od**
⟨every_s⟩	::=	**every** t **do** ⟨stmt⟩ **od**

2.3 Informal Semantics

We now provide a brief, informal semantics for these constructs. In section 5.9 we expand on the ideas presented here, and we also present the formal semantics for the language.

The **wait** statement specifies a pure delay for t time units, while **skip** is syntactic sugar for the construct **wait** 1. The read statement, $a?$, waits indefinitely for a communicating process to execute the corresponding write statement, or $a!$. The **exec**(a, m, n) construct denotes local computation – the event a may be executed for a minimum of m, and a maximum of n time units. Sequential composition is similar to that in the traditional, untimed CSP.

The guarded statement is a prioritized variant of that presented in [11]. In the version without a timeout, all of the communication guards delay indefinitely, waiting to be matched with their communicating partners. As soon as the first match is made, the guard with the highest priority takes precedence, and the statement associated with it is executed. However, note that we also allow local events as guards, and if one of these is included no delay is necessary. Thus the priority arbitration occurs immediately. And, if a timeout guard, wait t, is included in the statement, communication is only attempted for up to t time units, after which the timeout statement is executed.

The **interrupt** operator functions in the following manner: To be interrupted, the main body must currently be executing an event that has lower priority than the interrupting event. If this is the case, control transfers immediately to the interrupt handler. The **within** statement specifies that its body must execute within a specified time limit. If it fails to do so, an exception statement is executed. Note that this facility provides for the specification of nested temporal scopes [12], as **within** statements may themselves be nested. The **loop** statement specifies general, unguarded recursion, while the **every** construct denotes a statement that executes periodically.

There are two types of concurrent operators: Interleaving is denoted by the "&" symbol, while true parallelism is represented by the "||" symbol. True parallelism can take place only between different resources, while interleaved processes execute on the same resource. In fact, all expansions of the ⟨resource⟩ nonterminal are *required* to be executed on a single resource (or processor). This is guaranteed by the restrictions inherent in the grammar, and assumed in the construction of the formal semantics.

To a certain extent CSR provides not only a real-time programming paradigm, but also a configuration language. Unlike other CSP-influenced languages, the structure of our language *mandates* that process-to-resource mapping be performed. Processes are allocated to a single resource by simply expanding the ⟨resource⟩ nonterminal. When no additional process is to be added, the resource is closed. This is done by using the *close* operator, or "{ . }." And after a resource is closed, no other processes may compete for its allocation. At that time it is considered a resource, and can only be combined in parallel with other resources in the system.

There are several significant restrictions made on the events used both within and between resources. First, if an event a represents a synchronizing action, a single resource may not use a for both reading and writing. Recall that communication is one-to-one, *between* resources. If a resource uses both functions of the event it may communicate with itself. And since all actions on a single resource are purely interleaved, it is impossible for the read and write actions to occur simultaneously. Thus if interleaved processes need to communicate with each other, they must utilize intermediate resources such as memory, communication media and the like.

Next, two *different* resources may not model a common function using the same event. For example, given an event a, two different resources cannot execute the "$a!$" statement. This would also violate our restriction that all communication must be one-to-one. If many-to-one communication protocols are desired (as in Ada [18]), they must explicitly be modeled through guarded statements.

One final restriction is that no two resources may share a single local event. Again, a local event is considered a unique unit from a particular resource. Thus, sharing it would violate the very resource constraints that we are attempting to model.

To some readers, many of these restrictions may appear overly harsh. Superficially at least, it seems that two interleaved processes *should* be able to directly communicate with each other. Yet cost must be assessed where cost is due, and permitting two such simultaneous actions would lead to improper conclusions about the system. At the very least it should require one time unit for the sender to send, and another unit for the receiver to receive. In most operating systems this type of communication is performed by mailboxes or signals. Such mechanisms require time to execute.

It should be noted that our grammar excludes some of the constructs permitted by many other concurrent languages. For example we do not implicitly allow a simple fork-join program, such as

$$Q = P_1; (P_2 \,\|\, P_3); P_4$$

In a typical language this program can be written without regard to such details as resource allocation, control between processors and the like. Yet if P_2 and P_3 are to be on separate resources, we cannot assume that they both begin *exactly* when P_1 ends. It is even more unlikely that P_4 will start exactly when either P_2 or P_3 ends, whichever is slowest. Indeed, at both the fork and join transitions there is always "hidden" resource consumption, such as operating system overhead. To analyze the correct temporal behavior of such a system, this type of resource consumption must be explicitly modeled.

3 Extended CSR

The "pure" CSR language described above captures priority-based interleaving subject to resource availability, and pure parallelism subject to event synchronization. The language is quite powerful, and can successfully model a real-time system consisting of shared resources. However, the modeling of communication through instantaneous, synchronizing events becomes too cumbersome a task when describing large, real-time applications. For example, two processes sharing a single resource cannot directly synchronize with each other; they must communicate through an intermediate resource, such as memory.

In this section, we augment the basic language with the notion of *channels* (or communication ports), and provide asynchronous send and receive operations on them. While the communication media must still be explicitly modeled, we keep this layer transparent to the application processes. The processes may communicate with each other in a homogeneous manner, regardless of the various resources they use.

The major extension to the language is in the expanded definition of the ⟨process⟩ nonterminal:

```
       ⟨process⟩  ::=  ⟨process-header⟩⟨stmt⟩ |
                       ⟨process⟩ & ⟨process⟩
⟨process-header⟩  ::=  process ⟨ident⟩
                           [ input ⟨channel-defs⟩ ]
                           [ output ⟨channel-defs⟩ ]
                           [ local ⟨channel-defs⟩ ]
  ⟨channel-defs⟩  ::=  ⟨ident⟩ (⟨priority⟩) [, ⟨channel-defs⟩ ]
```

Each user process declares the communication channels that it is going to use for messages, along with their associated priorities: *input* channels are for receiving messages and *output* channels are for sending messages. Local computation events are declared in a similar manner, although they retain their previous flavor.

Two processes *asynchronously* communicate on channel c using two primitives, **a_send**(c) and **a_recv**(c). A process that invokes **a_send**(c) may execute its next statement as soon as the communication medium accepts a message on channel c. A process that invokes **a_recv**(c) is delayed until there is a message present on channel c. These primitives are expanded from the ⟨stmt⟩ nonterminal, and can be used wherever a statement may appear. In fact, they translate into simple read and write statements, as we shall show below.

In the following example, process P_1 receives messages from P_2 on channel c. Also, P_1 reserves the event a for local computation, while P_2 uses a local event b.

```
process P1                  process P2
    input c(10)                 output c(1)
    local a(1)                  local b(2)
        process body                process body
```

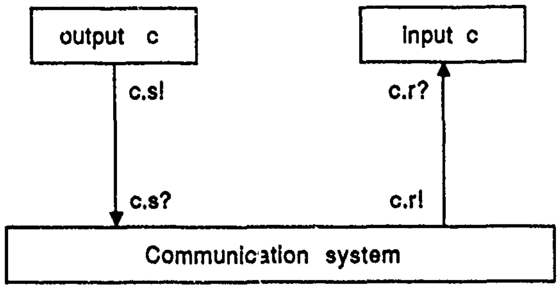

Figure 1: Translation of channels to events

The translation from channels to events is quite straightforward. In P_1, the channel c is simply translated to the event $c.r$, while in P_2, c is mapped to the event $c.s$ (see Figure 1). The headers of the two processes establish the following event priorities:

$$PRI_i(c.r) = 10 \qquad PRI_l(a) = 1 \qquad PRI_o(c.s) = 1 \qquad PRI_l(b) = 2$$

As for the translation of the communication primitives, any appearance of **a_recv**(c) in P_1 is simply replanced by the statement "$c.r?$". Similarly, the **a_send**(c) primitive in P_2 is replaced by "$c.s?$".

It is the responsibility of the communication medium to synchronize with $c.r$ and $c.s$. in a manner such that the asynchronous protocol is maintained. This underlying medium may be as simple as an interrupt controller, or as complex as a wide-area network. The communication protocol is highly application-dependent, and must be modeled separately for each real-time system.

4 An Example: Modeling a Distributed Robot System

Consider a robot system with a tactile sensor. The purpose of the system is to move the robot arm as determined by the controller until the arm touches an object. This distributed robot system consists of processors, a robot arm, a tactile sensor and a communication link. There are four processes: a controller, a robot arm driver, a robot and a tactile sensor.

4.1 Robot and Sensor Processes

Figure 2 shows communication dependencies between the four processes. There are six communication channels, two of which, *stopped* and *touched*, carry interrupts. The controller process sends the *next_p* to the robot arm driver. The robot arm driver interrupts the controller process when the arm has stopped through the *stopped* channel. Similarly, the sensor process sends a

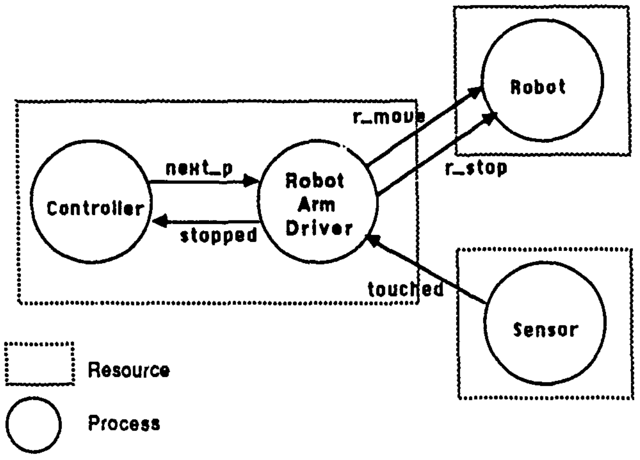

Figure 2: Communication Structure of the Distributed Robot System

touched interrupt to the robot arm driver when an object has been touched. The robot arm driver moves the arm by sending *r_move*, or stops it by sending *r_stop*.

Figure 3 details each process, written in extended CSR. Every 28 time units, the controller process determines a new arm position, which takes between 10 and 14 time units, and sends it to the arm driver process. This procedure continues until the robot arm driver notifies the controller process that the arm has stopped. Based on the current arm coordinates and the new position received from the controller, the robot arm driver computes the joint angles. This computation requires between 5 and 8 time units, after which the robot arm is moved. A sensor is attached to the arm, and every 10 time units the sensor process determines whether an object has been touched. If such a determination is made, the sensor process informs the robot arm driver, which must stop the robot as soon as possible. Thereafter, the arm driver process notifies the controller process. For simplicity, the sensor process is modeled as choosing nondeterministically between **skip** and **a_send**(*touched*).

4.2 A Complete Robot System

Since the timing behavior of the robot system depends not only on synchronization between processes but also on resource availability, it is necessary to know which processes are competing for the same resource. Suppose that the robot system is to run on three processors connected by a shared communication link such as an Ethernet. The controller and robot arm processes are assigned on one processor and the robot and sensor processes are their on own processors. Thus, channels *next_p* and *stopped* are for local communication, whereas the other channels are for non-local communication. Figure 4 shows the complete robot system written in CSR using the translations described in Section 3. For example, channel *next_p* is modeled using

```
process Controller
    output next_p(1)
    input stopped(2)
    local c(1) /* compute next position */
    interrupt on stopped do
        every 28 do exec(c,10,14); a_send(next_p) od
        when a_recv(stopped) → skip /* task completed */
    od
process RobotArm
    input next_p(2), touched(3)
    output r_move(2), r_stop(2), stopped(2)
    local a(2) /* compute joint angles */
    interrupt on touched do
        loop do a_recv(next_p); exec(a,5,8); a_send(r_move) od
        when a_recv(touched) → exec(a,5,8); a_send(r_stop); a_send(stopped)
    od
process Robot
    input r_move(1), r_stop(2)
    loop
        [ a_recv(r_move) → skip □ a_recv(r_stop) → skip ]
    od
process Sensor
    output touched(2)
    local sense(1)
    every 10 do
        [ sense → skip □ sense → a_send(touched) ]
    od
```

Figure 3: Distributed Robot System Written in the Extended CSR

```
   System    =  Node₁ || Node₂ || Node₃ || Network
    Node₁    =  { Controller & RobotArm } || LCS
    Node₂    =  { Sensor }
    Node₃    =  { Robot }
      LCS    =  { next_p.s?; next_p.r! || stopped.s?; stopped.r? }
  Network    =  { [ touched.s? → touched.r! □ r_move.s? → r_move.r! □ r_stop.s? → r_stop.s!] }
Controller   =  interrupt on stopped.r do
                    every 28 do exec(c,10,14); next_p.s! od
                    when stopped.r? → skip
                od
 RobotArm    =  interrupt on touched.r do
                    loop do next_p.r?; exec(a,5,8); r_move.s! od
                    when touched.r? → exec(a,5,8); r_stop.s!; stopped.s!
                od
    Robot    =  loop [ r_move.r? → skip □ r_stop.r? → skip ] od
   Sensor    =  every 10 do [ sense → skip □ sense → touched.s! ] od
```

Figure 4: Complete Robot System Written in CSR

two events, *next_p.s* and it next_p.r. Processes *LCS* and *Network* provide local communications within Node$_1$ and non-local communications. Nodes 1, 2 and 3 are closed to form resources since no additional processes are going to be assigned to them. We note that the closing of the *Network* process does not change its meaning, since it does not contain any local events.

5 A Denotational Semantics for CSR

[illegible]

The semantics of our language is based, in large part, on the linear-history paradigm first presented in [3]. Updated to model a real-time variant of CSP in [11], it was further revised and provided with a corresponding operational semantics in [7]. All three of these models have contributed to the formulation of our semantics.

The two abovementioned real-time models subscribed to the "maximum parallelism" view of concurrency, which was presented in [17]. Briefly, maximum parallelism implies that within any given process, delay is kept to a minimum. Or, at any given time instant, if a process is able to communicate whenever its partner is ready, it will communicate. Thus "pure" maximum parallelism implies that the computational resources are unlimited, or at least every process is mapped to its own processor. There is no interleaving *per se* – only the guarded command can model competition for resources.

Furthermore, the "pure" maximum parallelism implies that a static, bi-level priority scheme underlies the computational model: competition between idling and execution is always resolved in favor of the latter. Thus "ties" between simultaneous events are always arbitrated nondeterministically. For example, consider the system $S = R_1 \| R_2 \| R_3$, where

$$R_1 = \{[a? \rightarrow b? \,\Box\, b? \rightarrow a?]\} \quad R_2 = \{a!\} \quad R_3 = \{b!\}$$

Under maximum parallelism without priorities, we would reason that either event a is communicated first, and then b; or, b is communicated first, and then a. The decision between these alternatives would be "resolved" nondeterministically. With the use of priorities, our language allows us eliminate this nondeterminism if desired: thus, if $PRI_i(a) > PRI_i(b)$, the first alternative would always hold.

5.1 States

As in [11, 7], the execution of a process is represented by a collection of $\langle state, history\rangle$ pairs. The state component is used merely to depict whether a computation has finished. The two-valued state domain is denoted $\mathcal{S} = \{\bot, \top\}$, where $\bot$ is associated with an incomplete computation, and $\top$ denotes that a computation is complete. Thus the state $\top$ corresponds to the $\surd$ element in most trace and failure-based models [1, 5, 6, 16]. We let the symbol σ range over $\mathcal{S}$.

5.2 Histories

A history records a program's behavior over a certain period of time. For example, if the history has a length of i elements, the recorded period of activity is i time units. The i-th element represents a possible activity at time i.

Each element in a history is called an *assumption record*, which is pair $(A, \Delta) \in \mathcal{P}(\Sigma) \times \mathcal{P}(\mathcal{P}(\Sigma))$. Since the semantics captures true concurrency, at every time unit there may be a set of events that executes. If an assumption record appears as the i-th element in the history, the events in the A component may execute at time i. The Δ component contains other sets of events that also may execute at time i; however, these sets all have an equal or higher priority than that of A. The parallel operator ensures that if an $A' \in \Delta$ can synchronize with its communicating events, and if the priority of A' is higher than that of A, the set A will not be executed.

For example, examine the R_1 fragment in section 5, with $PRI_i(a) > PRI_i(b)$. There are three assumption records that describe the possible behaviors at the first time unit:

$$1.\ (\emptyset, \{\{a\}, \{b\}\}) \qquad 2.\ (\{b\}, \{\{a\}\}) \qquad 3.\ (\{a\}, \emptyset)$$

Record 1 shows the possibility of neither a nor b being communicated at time 1, and thus, the processor may idle. However it also shows that if either event is communicated the record will be deleted, as both events have a higher priority than idling. Record 2 shows that the event b may be communicated, but it also shows that competition between b and a is resolved in favor of a. Thus if a "tie" between a and b occurs, this record is deleted. Record 3 shows that the event a may be communicated at time 1, and also that it "wins" competitions for its processor.

We let the letters r, s and t range over assumption records. We define two selectors on assumption records. Letting $r = (A, \Delta)$ we define $f_1(r) = A$ and $f_2(r) = \Delta$.

As we have stated, the information captured in each assumption record is valid for one time unit. A real-time behavior of a program is captured by a *history*, or a sequence of these records. Histories are denoted by the domain $\mathcal{H} = \{r \mid r \in \mathcal{P}(\Sigma) \times \mathcal{P}(\mathcal{P}(\Sigma))\}^*$. We let the letter h range over the domain of histories, and in the spirit of [7], we use the following notation for them:

- $h_1 \hat{}\, h_2$ represents the concatenation of h_1 and h_2.
- λ is the empty history, where $\lambda \hat{}\, h = h \hat{}\, \lambda = h$.
- r^n is the history formed by n concatenations of the record r.
- $|h|$ is the length of the history h.
- $h[i]$ represents the i-th record of the history h. If $i > |h|$, define $h[i]$ to be $(\emptyset, \emptyset)$. For any history h, $h[0] = \lambda$.
- $h_1 < h_2$ iff $\exists h \in \mathcal{H}, h \neq \lambda \,.\, (h_1 \hat{}\, h = h_2)$.

5.3 Partial Ordering, Prefix Closure and Fixed Points

We denote our domain of linear histories as $\mathcal{SH} = \mathcal{S} \times \mathcal{H}$, and we let the letter X range over it. The domain forms a complete partial order. First, $\langle \perp, \lambda \rangle$ is the least element. The ordering is formed as follows. If $X_1 \subseteq \mathcal{SH}$ and $X_2 \subseteq \mathcal{SH}$, then $X_1 \sqsubseteq X_2 \quad$ iff $\quad X_1 \subseteq X_2$. Thus, least upper bounds of chains are determined by taking the union of every set.

We require that the computations of every program are *prefix-closed*, which ensures that all of our operators are well-defined. That is, they all generate a least fixed point set of $\langle$*state*, *history*$\rangle$ pairs. For a given set $X \in \mathcal{SH}$, we denote the *prefix closure* of X as

$$_{\leq}X = X \cup \{\langle \perp, h' \rangle \mid \exists \langle \sigma, h \rangle \in X . h' < h\}$$

5.4 Sequential Composition of Sets in $\mathcal{SH}$

Assume that a statement S_1 can generate a set of behaviors X_1, and S_2 can generate a set of behaviors X_2, with both X_1 and X_2 in $\mathcal{SH}$. We often wish to sequentially compose the behaviors from two sets such as these. That is, if an element of X_1 is finished we may append an element of X_2. Formally, the sets are composed using the function $\mathcal{C} \in \mathcal{SH} \times \mathcal{SH} \rightarrow \mathcal{SH}$, where

$$\begin{aligned} \mathcal{C}(X_1, X_2) = \ & \{\langle \bot, h\rangle \mid \langle \bot, h\rangle \in X_1\} \cup \\ & \underline{\leq}\{\langle \sigma, h_1 \hat{\ } h_2\rangle \mid \langle \top, h_1\rangle \in X_1 \wedge \exists h_2 . \langle \sigma, h_2\rangle \in X_2\} \end{aligned}$$

5.5 Semantic Representation of Programs

A process [1] $P = (res, imp, exp, p, traces)$ is defined as follows:

1. $res \in \mathcal{P}(\Sigma)$, the set of events have been resolved in the process P. Whether originally used to model local computation or synchronization, these events are now resolved and are local to P.
2. $imp \in \mathcal{P}(\Sigma)$, the set of events that are imported, *i.e.*, those on which P may exercise as "read" actions.
3. $exp \in \mathcal{P}(\Sigma)$, the set of events that are exported, *i.e.*, those on which P may exercise as "write" actions.
4. $p \in \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \rightarrow BOOL$, the priority function for P. For two sets of events A and B, if the predicate $p(A, B)$ holds, we say that the "priority of A is less than or equal to the priority of B."
5. $traces \in \mathcal{SH}$, the set of potential executions of the process P.

For convenience, we make the following definitions:

$$\rho(P) = res \quad \iota(P) = imp \quad \epsilon(P) = exp \quad \pi(P) = p \quad \tau(P) = traces$$

In addition, we occasionally refer to the *alphabet* of a process, or $\alpha(P)$, as its complete set of observable events. That is, the alphabet of a process is the union of its resolved, imported and exported events: $\alpha(P) = \rho(P) \cup \iota(P) \cup \epsilon(P)$.

[1] Here, unless ambiguous, we call the semantic representations of *all* program fragments processes. We use this terminology whether the *syntax* of the fragment is a statement, process, resource or system, as defined by the grammar.

5.6 Event Consistency

In section 2.3 we discussed some constraints placed on the events in a program's substructures; now we can treat them formally. Assume that $P_1, P_2, \ldots, P_n$ are processes being combined to run on the same resource; that is, their syntax was expanded from a single ⟨resource⟩ nonterminal. The following constraints must hold:

$$\begin{aligned}
Consistent_{\&} = \; & \forall i \forall j \\
& \iota(P_i) \cap \epsilon(P_j) = \emptyset \wedge \epsilon(P_i) \cap \iota(P_j) = \emptyset \wedge & (1) \\
& (\epsilon(P_i) \cup \iota(P_i)) \cap \rho(P_j) = \emptyset \wedge (\epsilon(P_j) \cup \iota(P_j)) \cap \rho(P_i) = \emptyset & (2)
\end{aligned}$$

Line (1) enforces that no two processes, running on the same processor, can instantaneously communicate with each other. Since the two processes have their executions interleaved, such behavior would be impossible. Line (2) enforces that local computation and communication cannot be modeled by the same event. Local computation requires no synchronization with external resources, while communication does.

Now assume that P_1 and P_2 are subsystems consisting of closed resources, to be combined by the "||" operator. We insist that the following constraints must hold:

$$\begin{aligned}
Consistent_{||} = \; & \iota(P_1) \cap \iota(P_2) = \emptyset \wedge \epsilon(P_1) \cap \epsilon(P_2) = \emptyset \wedge & (1) \\
& \rho(P_1) \cap \rho(P_2) = \emptyset \wedge & (2) \\
& (\iota(P_1) \cup \epsilon(P_1)) \cap \rho(P_2) = \emptyset \wedge (\iota(P_2) \cup \epsilon(P_2)) \cap \rho(P_1) = \emptyset & (3)
\end{aligned}$$

Line (1) enforces that communication between resources is strictly one-to-one, with a single receiver and a single sender. Line (2) mandates that isolated resources are not shared: the local units of computation from each are private. Line (3) enforces that a single event cannot model both local computation and communication.

If the program fragments to be combined satisfy such constraints, we call them *consistent*. To avoid redundancy in our operator definitions, we assume that all fragments combined are, indeed, consistent.

5.7 Defining the Priority Functions

We use two functions to define the priority predicates for our combinators. One is used for creating processes that execute on a single resource, while the other is employed strictly in the definition of parallel composition. Let $P_1, P_2, \ldots, P_n$ be a group of processes, all of which are to run on the same resource. We assume that these processes are consistent, as defined in the previous section. If we let $res = \bigcup_{i=1}^{n} \rho(P_i)$, $imp = \bigcup_{i=1}^{n} \iota(P_i)$, and $exp = \bigcup_{1}^{n} \epsilon(P_i)$, then $Compose_{\&}(res, imp, exp)$ completely defines the priority function for this group of processes, where:

$$\forall A \subseteq \Sigma \forall B \subseteq \Sigma,\ Compose_{\&}(r,i,e)(A,B) = P_{\&}(r,i,e)(A) \leq P_{\&}(r,i,e)(B),$$

where

$$P_{\&}(r,i,e)(C) = \begin{cases} PRI_l(a) & \text{if } C \cap r = \{a\} \\ PRI_i(a) & \text{if } C \cap i = \{a\} \\ PRI_o(a) & \text{if } C \cap e = \{a\} \\ 0 & \text{otherwise} \end{cases}$$

The definition requires some explanation. First, if a group of processes are to run on a single resource, together they can only execute one event at a time. If two events are simultaneously offered from communicating partners, one *must* be rejected. Thus only singleton sets can have a nonzero priority. Note also that since the processes are assumed to be consistent, the function $P_{\&}(C)$ is well-defined.

When composing two truly concurrent systems with the "$||$" operator, the priority functions can be composed as follows: $\pi(P_1||P_2) = Compose_{||}(\pi(P_1), \pi(P_2))$, where

$$\forall A \subseteq \Sigma \forall B \subseteq \Sigma,\ Compose_{||}(p_1,p_2)(A,B) = p_1(A,B) \land p_2(A,B)$$

Let $p \in \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma) \rightarrow BOOL$ be a priority function, and let A and B be sets of events. For convenience we use the following notation:

$$A \leq_p B \text{ iff } p(A,B)$$
$$A =_p B \text{ iff } A \leq_p B \land B \leq_p A$$
$$A <_p B \text{ iff } A \leq_p B \land B \not\leq_p A$$

We construct our semantics to exploit this ordering. In particular, our parallel operator guarantees that if there is a choice between executing A and B, the selection is made by this priority ordering.

5.8 Assumption Records, Histories and Priorities

Now we can integrate the concepts of assumption records and process priorities. Let $P = (res, imp, exp, p, traces)$ be a process, and assume that $\langle \sigma, h\hat{\ }(A,\Delta)\rangle \in traces$. That is, h concatenated with the record (A,Δ) is a history of the process' behavior. First, by the construction of the assumption record, $\forall A' \in \Delta,\ A \leq_p A'$. This implies that for every $A' \in \Delta$, $h\hat{\ }(A',\Delta')$ is also a history of the process, where

$$\Delta' = \{B \in \Delta \cup \{A\} \mid B \neq A' \land A' \leq_p B\}.$$

Now assume that $A' \in \Delta$, with $A' >_p A$. Thus there is some pair $\langle \sigma', h\char`^(A', \Delta')\rangle$ in *traces*. However, it cannot be said that the events in A' *will* be active rather than those in A. If some event is used for communication, another participating process is needed for synchronization. Thus, only when all of the communications in A' are resolved can we say that A' *will* be selected rather than A. When this occurs we delete the history $h\hat{}(A, \Delta)$, as it will never be observed.

To formalize this concept, once a processor has been closed, we maintain that all traces are *prioritized*, that is: $\forall\langle\sigma, h\rangle \in traces$, $prioritized(p, h, imp \cup exp)$, where

$$prioritized(p,h,B) \quad \text{iff} \quad \forall i \leq |h| \; \forall A' \in f_2(h[i]), \;\; f_1(h[i]) <_p A' \Longrightarrow A' \cap B \neq \emptyset$$

In other words, assume that $h_1\hat{}(A, \Delta)$ is some history of a process P_1. If there is a set $A' \in \Delta$ with $A <_p A'$, we guarantee that A' contains at least one unresolved event from P_1's alphabet. That is, in order for all the events in A' to execute, P_1 must be composed in parallel with some other process. On the contrary, assume that A' contained only local events. In this case it would execute instead of A, and thus $h_1\hat{}(A, \Delta)$ would not be a history of P_1.

5.9 The Meaning Function

In this section we develop a meaning function "$[\![.]\!]$", which maps the syntax of CSR to the domain of semantic processes. Often we make use of the *partition* function, defined as follows:

$$partition(p, A, \Delta) = (A, \; \{A' \in \Delta \cup \{\emptyset\} \mid A \neq A' \wedge A \leq_p A'\})$$

Here Δ may contain event sets with lower priority than that of A; they are filtered out of the record returned by the function. We note here that $\emptyset$ represents an idling behavior, and for any priority function p, $\emptyset$ has the lowest priority in the partial order defined by p.

5.9.1 Wait

The **wait** statement specifies an idle period of exactly t time units, where $t \geq 1$. As can be seen from the associated priority function p, idling has the lowest priority.

$$[\![\textbf{wait } t]\!] = (\emptyset, \emptyset, \emptyset, Compose_{\&}(\emptyset, \emptyset, \emptyset), \leq\{\langle \top, (\emptyset, \emptyset)^t\rangle\})$$

5.9.2 Skip

The **skip** statement is syntactic sugar for **wait** 1; thus **skip** requires 1 time unit to execute:

$$[\![\textbf{skip}]\!] = [\![\textbf{wait } 1]\!]$$

5.9.3 Write

The *write* construct declares that a resource is *ready* to synchronize on an exported event. The statement delays indefinitely until the communication is successful, that is, when the sending resource issues a corresponding "read" event. The time that this occurs depends on 1) the priority of the write event with respect to other events competing for its resource, 2) the time that the corresponding read event becomes ready, and 3) the priority of the read event with respect to other events competing for *its* resource.

$$[\![a!]\!] = (\emptyset, \emptyset, \{a\}, p, traces)$$

where

$$p = Compose_{\&}(\emptyset, \emptyset, \{a\})$$

$$traces = {\le}\{\langle \top, (\emptyset, \{\{a\}\})^{i \frown} partition(p, \{a\}, \{\emptyset\})\rangle \mid i \ge 0\}$$

The first part of the history, or $(\emptyset, \{\{a\}\})^i$, denotes that after a becomes ready, there may be an indefinite idling period before the communication is successful. But the presence of the set $\{a\}$ shows that throughout this idling period, the event a remains ready. The second part of the history is composed of a single assumption record: $partition(p, \{a\}, \{\emptyset\})$, which represents the success of the communication. This record can have two possible values, depending on the priority function. If $PRI_o(a) > 0$, the record's value is $(\{a\}, \emptyset)$, while if $PRI_o(a) = 0$, the record becomes $(\{a\}, \{\emptyset\})$. In the first case, the processor cannot arbitrarily delay itself when the matching read event becomes available. In the second case delays may be inserted, which assigns the choice to communicate exclusively to the receiver.

5.9.4 Read

The *read* statement is the exact dual of its *write* counterpart explained in section 5.9.3. Here a is placed in the import alphabet; the *write* statement contains a in its export alphabet. Also, the the priority value for a is taken from the PRI_i function.

$$[\![a?]\!] = (\emptyset, \{a\}, \emptyset, p, traces)$$

where

$$p = Compose_{\&}(\emptyset, \{a\}, \emptyset)$$

$$traces = {\le}\{\langle \top, (\emptyset, \{\{a\}\})^{i \frown} partition(p, \{a\}, \{\emptyset\})\rangle \mid i \ge 0\}$$

5.9.5 Exec

The **exec** statement specifies that local computation events must compete for processor time. The exact number of time units required may be nondeterministic – thus we allow a range of time units to be specified. Here, m is the lower bound on the number of executed events, and n is the upper bound. It is assumed that $1 \leq m \leq n$.

$$[\![\mathbf{exec}(a, m, n)]\!] = (\{a\}, \emptyset, \emptyset, p, \mathit{traces})$$

where

$$p = \mathit{Compose}_{\&}(\{a\}, \emptyset, \emptyset)$$

$$\mathit{traces} = \textstyle\bigcup_{i=m}^{n} P^i(\langle \top, \lambda \rangle)$$

with

$$P(X) = \mathcal{C}(\leq\{\langle \top, (\emptyset, \{\{a\}\})^j \hat{\ } \mathit{partition}(p, \{a\}, \{\emptyset\}) \rangle \mid j \geq 0\}, X)$$

Note that here we make use of the trace composition operator, or "$\mathcal{C}$." In effect we compose between m and n individual executions of the event a, and we include all of the "ready" assumption records that precede each execution. Although the execution time of a does not depend on any communicating partner, it heavily depends on the priorities of the other events contesting for processor time. To make the interleaving combinator associative (see section 5.9.12), we must provide the possible "gaps" that exist between each local execution. They may, of course, be occupied by other local events of a higher priority. Only when the resource is closed can the unnecessary gaps be eliminated.

5.9.6 Guarded Choice

Our guarded choice construct is priority-based version of those presented in [11] and [7]. Each of the guards g_i may be one of the following: 1) a read guard, "$a?$", 2) a write guard, "$a!$", or 3) a local execution guard, "a". The time associated with the wait guard is assumed to be greater than 0. If no wait guard is present, we assume that the value of t to be infinite. As in related languages, the execution of a guard g_i is immediately followed by that of S_i.

An event-based guard must be executed within the specified t time units; if not, S_{n+1} is executed. An event-based guard may be delayed for one of two reasons. In the case of a local event, there may be contention for local processor time; that is, interleaved. higher-priority events are given the "right of way." The communicating guards also suffer from local competition; moreover, they must wait for their corresponding communicating partners to be successfully executed. It should be noted that these partners are also affected by the local competition on *their* processors. Thus, we can begin to see one of the ramifications of local competition for resources: although two processes may be willing to synchronize at a given time, actual communication may be delayed due to local resource contention. This is a radical departure from the "pure" maximal parallelism model.

$$[\![[g_1 \to S_1 \Box \ldots \Box g_n \to S_n \,\triangle\, \mathbf{wait}\, t \to S_{n+1}]]\!] = (res, imp, exp, p, traces),$$

where

$$res = \textstyle\bigcup_{i=1}^{n} \rho([\![g_i]\!]) \cup \bigcup_{i=1}^{n+1} \rho([\![S_i]\!])$$

$$imp = \textstyle\bigcup_{i=1}^{n} \iota([\![g_i]\!]) \cup \bigcup_{i=1}^{n+1} \iota([\![S_i]\!])$$

$$exp = \textstyle\bigcup_{i=1}^{n} \epsilon([\![g_i]\!]) \cup \bigcup_{i=1}^{n+1} \epsilon([\![S_i]\!])$$

$$p = Compose_{\&}(res, imp, exp)$$

$$traces =$$

$$\leq\{\langle \sigma, h_1\hat{\ }h_2\hat{\ }h_3\rangle \mid h_1 \in W \wedge |h_1| < t \wedge \exists i \leq n \,.\, \langle \sigma, h_3\rangle \in \tau([\![S_i]\!]) \wedge$$
$$h_2 = partition(p, \alpha([\![g_i]\!]), \{\alpha([\![g_j]\!]) \mid 1 \leq j \leq n\})\} \cup$$
$$\leq\{\langle \sigma, h_1\hat{\ }h_2\rangle \mid h_1 \in W \wedge |h_1| = t \wedge \langle \sigma, h_2\rangle \in \tau([\![S_{n+1}]\!])\}$$

where

$$W = \{(\emptyset, \{\alpha([\![g_1]\!]), \ldots, \alpha([\![g_1]\!])\})^n \mid i \geq 0\}$$

Example 5.1 This example depicts the interaction between prioritized events. Here a and b are both imported communication events, with $PRI_i(a) = 1$ and $PRI_i(b) = 2$:

$$S = [a? \to b? \,\Box\, b? \to a?]$$

Informally, the semantics for S can be explained as: "Wait for either event a or event b to be received. If event a is received first, accept it and wait for b. If event b is received first, accept it and wait for a. If both events are received simultaneously, accept event b and wait for a." The following are the *traces* of S, as computed by the definition of guarded choice:

$$\leq\{\langle \top, (\emptyset, \{\{a\},\{b\}\})^{i}\hat{\ }(\{a\},\{\{b\}\})\hat{\ }(\emptyset,\{\{b\}\})^{j}\hat{\ }(\{b\},\emptyset)\rangle \mid i,j \geq 0\} \cup$$
$$\leq\{\langle \top, (\emptyset, \{\{a\},\{b\}\})^{i}\hat{\ }(\{b\},\emptyset)\hat{\ }(\emptyset,\{\{a\}\})^{j}\hat{\ }(\{a\},\emptyset)\rangle \mid i,j \geq 0\}$$

How is the priority structure reflected in the histories shown here? There is a major difference between the assumption records representing the two communicating guards. The record denoting successful communication with a is $(\{a\},\{\{b\}\})$. This shows that if event b is received simultaneously with a, the processor defers to b, and the history is removed. On the other hand, the record representing successful communication with b is $(\{b\},\emptyset)$; that is, no alternative of a higher priority exists. □

5.9.7 Sequential Composition

Sequential composition operates in the usual manner: traces from S_2 are appended to completed traces from S_1. Fortunately our trace composition operator, "$\mathcal{C}$", does just that; thus, the definition is straightforward:

$$[\![S_1;S_2]\!] = (res, imp, exp, p, traces)$$

where

$$res = \rho([\![S_1]\!]) \cup \rho([\![S_2]\!]), \quad imp = \iota([\![S_1]\!]) \cup \iota([\![S_2]\!]), \quad exp = \epsilon([\![S_1]\!]) \cup \epsilon([\![S_2]\!]),$$

$$p = Compose_{\&}(res, imp, exp)$$

$$traces = C(\tau([\![S_1]\!]), \tau([\![S_2]\!]))$$

5.9.8 Within

The **within** operator denotes that S_1 is to be initiated immediately, and that it must be completed within the specified t time units. If S_1 does not finish executing by time t, control is transferred directly to S_2. Thus S_2 can be considered a timeout exception handler of sorts. which is an essential construct in many real-time programs.

$$[\![\textbf{within } t \textbf{ do } S_1 \textbf{ when } t \rightarrow S_2 \textbf{ od}]\!] =$$

$$(\rho([\![S_1;S_2]\!]), \iota([\![S_1;S_2]\!]), \epsilon([\![S_1;S_2]\!]), \pi([\![S_1;S_2]\!]), traces)$$

where $traces =$

$$\leq\{\langle\top, h\rangle \mid \langle\top, h\rangle \in \tau([\![S_1]\!]) \wedge |h| \leq t\} \cup$$

$$\leq\{\langle\sigma, h_1{}^\frown h_2\rangle \mid \langle\bot, h_1\rangle \in \tau([\![S_1]\!]) \wedge \langle\sigma, h_2\rangle \in \tau([\![S_2]\!]) \wedge |h_1| \doteq t\}$$

Example 5.2 As an example, consider the following fragment:

$$S = \textbf{within } 10 \textbf{ do } a?; b? \textbf{ when } 10 \rightarrow \textbf{exec}(c, 1, 1) \textbf{ od}$$

Here, S must receive events a and b within 10 time units; otherwise it will attempt to execute the local event c. Thus if the exception handler is required, we can reason that the construct "$a?;b?$" did not complete one, or both of the communications. This can be seen from the *traces* of S (where we assume that $PRI_i(a)$, $PRI_i(b)$ and $PRI_l(c)$ are all greater than 0).

$$\leq\{\langle\top, (\emptyset, \{\{a\}\})^{i\frown}(\{a\}, \emptyset)^\frown(\emptyset, \{\{b\}\})^{j\frown}(\{b\}, \emptyset)\rangle \mid i + j \leq 8\} \cup$$

$$\leq\{\langle\top, (\emptyset, \{\{a\}\})^{10\frown}(\emptyset, \{\{c\}\})^{i\frown}(\{c\}, \emptyset)\rangle \mid i \geq 0\} \cup$$

$$\leq\{\langle\top, (\emptyset, \{\{a\}\})^{i\frown}(\{a\}, \emptyset)^\frown(\emptyset, \{\{b\}\})^{j\frown}(\emptyset, \{\{c\}\})^{k\frown}(\{c\}, \emptyset)\rangle \mid i + j = 9 \wedge k \geq 0\}$$

□

5.9.9 Interrupt

The **interrupt** operator takes full advantage of our priority semantics, and we have found it invaluable in our specifications of robot-sensor systems [4]. The construct initiates S_1, which will be interrupted only if 1) the imported event a is detected, and 2) the event executing in S_1 has a lower priority than $PRI_i(a)$. If both of these conditions are met, S_1 is immediately killed, a is received, and control is transferred to the interrupt-handler, S_2. This is not the more common type of interrupt construct, where control would be transferred back to S_1 at the conclusion of S_2. Interrupt handlers of that nature can be specified using the *interleave* operator (see section 5.9.12).

$$
\begin{aligned}
&[\![\textbf{interrupt on } a \textbf{ do } S_1 \textbf{ when } a? \rightarrow S_2 \textbf{ od}]\!] = \\
&\quad (\rho([\![S_1]\!]) \cup \rho([\![S_2]\!]), \iota([\![S_1]\!]) \cup \iota([\![S_2]\!]) \cup \{a\}, \epsilon([\![S_1]\!]) \cup \epsilon([\![S_2]\!]), p, traces) \\
&\text{where} \\
&\quad p = Compose_{\&}(\rho([\![S_1]\!]) \cup \rho([\![S_2]\!]), \iota([\![S_1]\!]) \cup \iota([\![S_2]\!]) \cup \{a\}, \epsilon([\![S_1]\!]) \cup \epsilon([\![S_2]\!])) \\
&\quad traces = \\
&\qquad \underline{\prec}\{\langle \top, I(h)\rangle \mid \exists \langle \top, h\rangle \in \tau([\![S_1]\!])\} \cup \\
&\qquad \underline{\prec}\{\langle \sigma_2, h\,\hat{}\,h_2\rangle \mid \langle \sigma_2, h_2\rangle \in \tau([\![S_2]\!]) \wedge \exists \langle \bot, h_1\hat{}(A, \Delta)\rangle \in \tau([\![S_1]\!]) . \\
&\qquad\qquad h = I(h_1)\hat{}partition(p, \{a\}, \{A\} \cup \Delta) \\
&\text{where} \\
&\quad I(h) = \{h' \mid \forall i \geq 1 \, \exists A \, \exists \Delta . \\
&\qquad h[i] = (A, \Delta) \wedge h'[i] = partition(p, A, \Delta \cup \{\{a\}\})\} .
\end{aligned}
$$

While somewhat complicated, the above definition for *traces* is quite easy to understand. First, the function $I(h)$ constructs a new history from h in the following manner. For every assumption record (A, Δ) in h, if $A \leq_p \{a\}$ then $\{a\}$ is inserted into Δ. Otherwise the record remains as (A, Δ). Thus $I(h)$ represents when h may be interrupted by a. So, the first set in the *traces* definition contains the original traces of S_1, plus this potential interrupt information.

The second set contains the traces showing where S_1 is interrupted. Each trace contains three parts, the first of which being the uninterrupted part. The second part consists of single record, depicting the time at which S_1 is interrupted. The third and final part is a history from the interrupt handler, S_2.

5.9.10 Loop

The loop construct is our representation of general recursion. Its continuity is contingent on the continuity of the trace composition operator "$\mathcal{C}$," a proof for which can be found in [11].

$$
\begin{aligned}
&[\![\text{loop do } S \text{ od}]\!] = (\rho([\![S]\!]), \iota([\![S]\!]), \epsilon([\![S]\!]), \pi([\![S]\!]), traces) \\
&\text{where } traces = \textstyle\bigcup_{i>0} P^i(\langle \bot, \lambda \rangle) \\
&\text{with } P(X) = \mathcal{C}(\tau([\![S]\!]), X)
\end{aligned}
$$

Example 5.3 As an example of **loop** and **interrupt**, the following fragment continuously executes the local event a, while waiting to be interrupted by b, at which time c is locally executed.

$$
\begin{array}{l}
\textbf{interrupt on } b \textbf{ do} \\
\quad \textbf{loop do exec}(a,1,1) \textbf{ od} \\
\textbf{when } b? \rightarrow \textbf{exec}(c,1,1) \textbf{ od}
\end{array}
$$

□

5.9.11 Every

The **every** operator is used to specify periodic behaviors, and is a timed variant of general recursion. Assume that for in a given case S requires i time units to execute. If $i \leq t$ then S runs to completion, after which there is a delay of $t - i$ units before the body is restarted. On the other hand, if $i > t$, the history is interrupted after t time units, at which time S is immediately reinitiated.

$$
\begin{array}{l}
[\![\textbf{every } \mathrm{t} \textbf{ do } S \textbf{ od}]\!] = (\rho([\![S]\!]), \iota([\![S]\!]), \epsilon([\![S]\!]), 1, \pi([\![S]\!]), \mathit{traces}) \\
\text{where } \mathit{traces} = \bigcup_{i>0} P^i(\langle \bot, \lambda \rangle) \\
\text{with } P(X) = \mathcal{C}(Y, X) \\
\text{where} \\
\quad Y = \,\leq\!\{\langle \top, h \rangle \mid \exists \langle \sigma, h \rangle \in \tau([\![S; \mathit{wait}\ t]\!]) \,.\, |h| = t\}
\end{array}
$$

5.9.12 Interleave Operator

The *interleave* operator accepts two processes, P_1 and P_2, and interleaves their histories to execute on a single resource. This means that only one event, either from P_1 or P_2, may be executed during a single time period. The arbitration between the two processes is done according to the priorities of the events ready to execute, in the same manner as guarded choice.

Note the first line of the definition, where the composed state σ is determined. As intuition would have it, the resolved history is considered complete only if both constituent histories are complete; otherwise it is considered incomplete. Also, the *comparable* predicate ensures that if two incomplete histories are composed, they must be of equal length. If only one history is complete, the length of the complete history cannot exceed that of the incomplete history.

These rules ensure the associativity of the operator.

$$[\![P_1 \& P_2]\!] = (\rho([\![P_1; P_2]\!], \iota([\![P_1; P_2]\!], \epsilon([\![P_1; P_2]\!], p, traces)$$

where

$$p = \pi([\![P_1; P_2]\!])$$

$$traces =$$

$$\leq \{\langle \sigma, h\rangle \mid \exists \langle \sigma_1, h_1\rangle \in \tau([\![P_1]\!]) \exists \langle \sigma_2, h_2\rangle \in \tau([\![P_2]\!]) . (\sigma = \sigma_1 = \sigma_2 = \top) \vee (\sigma = \perp) \wedge$$
$$comparable(\sigma_1, h_1, \sigma_2, h_2) \wedge$$
$$\forall i \geq 1 \exists A_1, A_2, \Delta_1, \Delta_2 . h_1[i] = (A_1, \Delta_1) \wedge h_2[i] = (A_2, \Delta_2) \wedge$$
$$(h[i] = partition(p, A_1, \{A_2\} \cup \Delta_1 \cup \Delta_2) \vee h[i] = partition(p, A_2, \{A_1\} \cup \Delta_1 \cup \Delta_2))\}$$

where

$$comparable(\sigma_1, h_1, \sigma_2, h_2) \iff (\sigma_1 = \perp \Rightarrow |h_2| \leq |h_1|) \wedge (\sigma_2 = \perp \Rightarrow |h_1| \leq |h_2|)$$

Example 5.4 Now we can specify the type of interrupt handler that returns to an interrupted program. In the fragments P_1 and P_2 we assume that

$$PRI_i(a_1) = PRI_i(a_2) = PRI_l(b_1) = PRI_l(b_2) = 1, \text{ and}$$
$$PRI_i(a_3) = PRI_l(b_3) = 2$$

$P_1 =$ **loop do** $[a_1? \rightarrow$ **exec**$(b_1, 3, 10) \,\square\, a_2? \rightarrow$ **exec**$(b_2, 4, 6)]$ **od**	$P_2 =$ **loop do** $[a_3? \rightarrow$ **exec**$(b_3, 5, 5)]$ **od**

Now, in $P_1 \& P_2$ we can consider **exec**$(b_3, 5, 5)$ an interrupt service routine, executed as a critical section to handle the interrupt a_3. After the interrupt is handled, P_1 can resume execution until another interrupt is detected. □

5.9.13. Close Operator

As stated in section 2.3, the *close* operator ensures that a program's resources may no longer be shared. This implies that no further process combinators may be applied to the operand. In particular, the predicate "*prioritized*" eliminates all time "gaps" that were being preserved for future resource sharing.

$$[\![\{S\}]\!] = (\rho([\![S]\!]), \iota([\![S]\!]), \epsilon([\![S]\!]), \pi([\![S]\!]), traces)$$

where

$$traces = \leq \{\langle \sigma, h\rangle \mid \langle \sigma, h\rangle \in \tau([\![S]\!]) \wedge prioritized(\pi([\![S]\!]), h, \iota([\![S]\!])\epsilon([\![S]\!]))\}$$

Example 5.5 We can easily illustrate the meaning of *close* when we apply it to the statement S in example 5.2. The *traces* of $\{S\}$ are now:

$$\leq\{\langle\top,(\emptyset,\{\{a\}\})^{i}\hat{}(\{a\},\emptyset)\hat{}(\emptyset,\{\{b\}\})^{j}\hat{}(\{b\},\emptyset)\rangle \mid i+j\leq 8\}\cup$$
$$\leq\{\langle\top,(\emptyset,\{\{a\}\})^{10}\hat{}(\{c\},\emptyset)\rangle\}\cup$$
$$\leq\{\langle\top,(\emptyset,\{\{a\}\})^{i}\hat{}(\{a\},\emptyset)\hat{}(\emptyset,\{\{b\}\})^{j}\hat{}(\{c\},\emptyset)\rangle \mid i+j=9\}$$

In other words, we no longer have to wait for the execution of the local event c. Once it becomes ready, it can be immediately executed. □

5.9.14 Parallel Operator

Finally we come to the parallel operator, with which we can specify true concurrency. Each of the operator's arguments are assumed to be closed processors – incapable of sharing any more of their resources. At this point they interact only with external events. Note how the composed alphabet is formed. All external events through which the two processes communicate are resolved, and transformed into local events. At this point they can are considered no differently than other local computation events.

The state resolution is identical to that in the interleave operator – a resolved trace can be considered complete only if both contributing traces are complete. Also like the interleave operator, the *comparable* predicate is used to ensure consistency of history length (see section 5.9.12 for its definition).

If two histories are to be combined, they must be synchronized on the events through which the two processes communicate. For example, assume that $h_1 \in \tau([\![P_1]\!])$ and $h_2 \in \tau([\![P_2]\!])$. Also, assume that some event a is imported by P_1 and exported by P_2. If there is some i such that $a \in f_1(h_1[i])$ but $a \notin f_1(h_2[i])$, these two histories cannot be composed. This is because at time i, P_1 is ready to communicate, while P_2 is not.

$[\![P_1 \| P_2]\!] = (res, imp, exp, p, traces)$

where

$res = \rho([\![P_1]\!]) \cup \rho([\![P_2]\!]) \cup (\iota([\![P_1]\!]) \cap \epsilon([\![P_2]\!])) \cup (\epsilon([\![P_1]\!]) \cap \iota([\![P_2]\!]))$

$imp = ($[illegible]$\epsilon([\![P_2]\!])) \cup ($[illegible]$ - \epsilon([\![P_1]\!]))$

$exp = (\epsilon([\![P_1]\!]) - \iota([\![P_2]\!])) \cup (\epsilon([\![P_2]\!]) - \iota([\![P_1]\!]))$

$p = Compose_{\|}(\pi(P_1), \pi(P_2))$

$traces =$

$\leq\{\langle\sigma,h\rangle \mid \exists\langle\sigma_1,h_1\rangle \in \tau([\![P_1]\!])\, \exists\langle\sigma_2,h_2\rangle \in \tau([\![P_2]\!]) . (\sigma=\sigma_1=\sigma_2=\top) \vee (\sigma=\bot) \wedge$
$comparable(\sigma_1,h_1,\sigma_2,h_2) \wedge (\forall i \geq 1\, sync(f_1(h_1[i]), f_1(h_2[i]))) \wedge$
$(\forall i \geq 1\, h[i] = combine(h_1[i], h_2[i])) \wedge prioritized(p, h, imp \cup exp)\}$

where

$$
\begin{aligned}
&sync(A_1, A_2) \iff \forall a \in ((\iota\llbracket P_1\rrbracket \cap \epsilon\llbracket P_2\rrbracket) \cup (\epsilon\llbracket P_1\rrbracket \cap \iota\llbracket P_2\rrbracket)), (a \in A_1 \Leftrightarrow a \in A_2)\\
&combine((A_1, \Delta_1), (A_2, \Delta_2)) =\\
&\quad (A_1 \cup A_2,\\
&\quad\ \{A_1' \cup A_2' \mid sync(A_1', A_2') \wedge\\
&\quad\quad ((A_1' \in \Delta_1 \ \wedge\ A_2' \in (\{A_2\} \cup \Delta_2)) \vee (A_2' \in \Delta_2 \ \wedge\ A_1' \in (\{A_1\} \cup \Delta_1)))\})
\end{aligned}
$$

Example 5.6 Now we can complete example 5.1. Let $P = \{S\}$, or

$$P = \{[a? \rightarrow b? \,\square\, b? \rightarrow a?]\}$$

and

$$Q = \{a!\} \quad R = \{b!\}$$

Assume that $PRI_o(a) = PRI_o(b) = 0$; that is, we give the resource P the exclusive right to choose when the communication occurs. It follows that:

$$
\begin{aligned}
\tau(\llbracket Q\rrbracket) = &\quad \leq\{\langle \top, (\emptyset, \{\{a!\}\})^{i\frown}(\{a\}, \{\emptyset\})\rangle \mid i \geq 0\}\\
\tau(\llbracket R\rrbracket) = &\quad \leq\{\langle \top, (\emptyset, \{\{b!\}\})^{i\frown}(\{b\}, \{\emptyset\})\rangle \mid i \geq 0\}
\end{aligned}
$$

and that

$$\tau(\llbracket P\|Q\|R\rrbracket) = \leq\{\langle \top, (\{b\}, \{\emptyset\})^\frown(\{a\}, \{\emptyset\})\rangle\}$$

Since $PRI_i(b) > PRI_i(a)$, this is exactly what we would expect. $\square$

6 Conclusion

In this paper we have presented a real-time specification language called CSR. This formalism can be considered as a step toward unifying real-time specifications with their corresponding implementations. The CSR paradigm imposes that specifications be *resource-based*; a process must explicitly be declared as resident on a particular resource. Moreover, this resource may host other processes, as we have shown in our robot-sensor example (section 4). Thus we have departed from the "pure" maximum parallelism model, which assumes a one-to-one allocation of processors to processes. To arbitrate between processes competing for a single resource, we have incorporated a general, event-based priority scheme. With prioritized events we can model not only *occam*'s PRI ALT and PRI PAR constructs, but more versatile systems, in which processes continuously alter their priorities over time.

CSR can specify, and give precise meaning to some behaviors necessary in modern real-time systems. For example, the **within** statement, with its associated exception-handler, can specify a *temporal scope* as described in [12]. The **every** statement accurately captures the behavior of a periodic process. The two interrupt-handling mechanisms – one that returns to an interrupted program, the other that kills it – can help denote the relative criticality of received events. To formalize all of our constructs, we have provided set of denotational semantics that captures both CSR's interleaved resource-sharing, and the "pure" concurrency that occurs between resources.

Much future work remains to be performed. To facilitate automated reasoning about CSR, we wish to provide it with an operational semantics, fully abstract with respect to the denotational semantics presented here. We plan to do this with a labeled transition system, in the spirit of [7]. However, we are not yet certain how its complexity will be affected by our interleaved, priority-based model.

Also, we are currently investigating whether the CSR formalism can be mapped to a system of communicating, finite-state machines. This technique has been explored for the "pure" maximum parallelism model [14], but again, we are unsure how the additional complexity of CSR will affect such an endeavor. However, if some variant of CSR can be mapped to finite-state machines, state exploration techniques can be used to statically detect properties such as liveness and deadlock.

Finally, we are extending the CSR formalism to permit the specification of dynamic priority schemes. The current model is quite powerful, in that processes can continuously alter their *own* priorities over time. However, the main thrust of our research is directed toward analyzing scheduling behavior. A scheduler has the ability to dynamically alter the priorities of *other* processes, based on the state of the system. Thus, we are currently incorporating the semantics of variable states into CSR. With this inclusion we will be able to reason about dynamic priorities, and therefore, about the properties of real-time scheduling algorithms.

References

[1] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[2] R. Cleaveland and M. Hennessy. Priorities in Process Algebras. In *Proc. of IEEE Symposium on Logic in Computer Science*, 1988.

[3] N. Francez, D. Lehmann, and A. Pnueli. A Linear History Semantics for Distributed Programming. *Theoretical Computer Science*, 32:25–46, 1984.

[4] R. Gerber and I. Lee. The Formal Treatment of Priorities in Real-Time Computation. In *Proc. 6th IEEE Workshop on Real-Time Software and Operating Systems*, 1989.

[5] R. Gerth and A. Boucher. A Timed Failure Semantics for Extended Communicating Processes. In *Proceedings of ICALP '87, LNCS 267*, 1987.

[6] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[7] C. Huizing, R. Gerth, and W.P. de Roever. Full Abstraction of a Denotational Semantics for Real-time Concurrency. In *Proc.* 14th *ACM Symposium on Principles of Programming Languages*, pages 223–237, 1987.

[8] C. Huizing, R. Gerth, and W.P. de Roever. *Modelling Statecharts Behavior in a Fully Abstract Way.* Technical Report TR. CSN 88/7, Department of Mathematics and Computing Science, Eindhoven University of Technology, July 1988.

[9] R. Janicki and P. Lauer. On the Semantics of Priority Systems. In *Proc. of Int. Conf. on Parallel Processing*, 1988.

[10] M. Joseph and A. Goswami. What's 'Real' about Real-time Systems? In *IEEE Real-Time Systems Symposium*, 1988.

[11] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional Semantics for Real-Time Distributed Computing. In *Logic of Programs Workshop '85, LNCS 193*, 1985.

[12] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming. In *IEEE Real-Time Systems Symposium*, 1985.

[13] L.Y. Liu and R.K. Shyamasundar. RT-CDL: A Real Time Design Language and its Semantics. In *IFIP '89*, 1989.

[14] L.Y. Liu and R.K. Shyamasundar. Static Analysis of Real-Time Distributed Systems. In *Proc. Symposium of Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS 331*, 1988.

[15] Inmos Ltd. *Occam Programming Manual.* Prentice-Hall, Inc., 1987.

[16] G.M. Reed and A.W. Roscoe. Metric Spaces as Models for Real-Time Concurrency. In *Proceedings of Math. Found. of Computer Science, LNCS 298*, 1987.

[17] A. Salwicki and T. Müldner. On the Algorithmic Properties of Concurrent Programs. In *Proceedings of Logic of Programs, LNCS 125*, 1981.

[18] U.S. Department of Defense. Ada Programming Language. 1983. ANSI/MIL-STD-1815A-1983.

[19] A. Zwarico. *Timed Acceptance: An Algebra of Time Dependent Computing.* PhD thesis. Department of Computer and Information Science, University of Pennsylvania, 1988.