# Communicating Shared Resources: A Paradigm For Integrating Real-Time Specification And Implementation

MS-CIS-91-31
GRASP LAB 259

Insup Lee
Susan Davidson
Richard Gerber

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

April 1991

# Communicating Shared Resources: A Paradigm for Integrating Real-Time Specification and Implementation

Insup Lee, Susan Davidson, and Richard Gerber
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

March 15, 1991

**Abstract**

The timed behavior of distributed real-time systems can be specified using a formalism called Communicating Shared Resources, or CSR. The underlying computation model of CSR is resource-based in which multiple resources execute synchronously, while processes assigned to the same resource are interleaved according to their priorities. CSR bridges the gap between an abstract computation model and implementation environments, but is too complex to be treated as a process algebra. We therefore give a calculus for CSR (CCSR), that provides the ability to perform equivalence proofs by syntactic manipulation. We illustrate how a CSR specification can be translated into the CCSR formalism using a periodic timed producer-consumer example, and how a translated CSR specification can be shown correct using syntactic manipulations.

## INTRODUCTION

The goal of this research is to develop a formal framework for reasoning about the temporal properties of real-time systems. Such a framework includes an appropriate specification language, an abstract model

*This research was supported in part by ONR N000014-89-J-1131.
To appear in Proc. ONR Workshop on Foundations of Real-Time Computing Research to be published by Kluwer Pub., 1991.

of computation, a notion of equivalence of terms in the model, and tools for automating proofs of equivalence.

Since the timing behavior of a real-time system depends not only on delays due to process synchronization, but also on the availability of shared resources, the computation model must include a notion of resources and how they can be shared as well as a notion of processes and synchronization. These notions are partially addressed in real-time models and scheduling theory, but not adequately combined. While most real-time models capture delays due to process synchronization, they abstract out resource-specific details by assuming idealistic operating environments. On the other hand, while scheduling theory captures the notion of resources, it ignores the effect of process synchronization except for simple precedence relations between processes. Therefore, a contribution of the model we develop is two integrate these two notions.

To help bridge the gap between abstract computation models and implementation environments, we have developed a real-time language 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 an action consisting of only a single event or particle. However, a resource may host a set of many processes, and at every instant, any number of these processes may compete for its availability. That is, on a single resource, the actions of multiple processes must be interleaved; "true" parallelism may take place only *between* resources. To arbitrate between competing events, CSR employs a priority-ordering.

CSR syntactically resembles variants of real-time CSP found in [1] and [2]. However, it also has the capacity to specify many constructs commonly found in real-time systems, such as timeouts, deadlines, periodic processes, temporal scopes [3] and exception-handling. CSR also incorporates several of the features of a configuration language, in that processes must be explicitly assigned to the resources on which they reside. We have formalized our constructs using a priority-based denotational semantics, that gives precise meaning to CSR's real-time characteristics, its interleaved resource sharing, and the "pure" concurrency that occurs between resources.

CSR supports a natural, high-level description of real-time systems,

and its semantics captures the temporal properties of prioritized resource interaction. However, the CSR language is far too complex to be treated as a process algebra, and thus does not easily lend itself to an equational characterization. To remedy this, we have developed the Calculus for Communicating Shared Resources, or CCSR. Strongly influenced by SCCS [4, 5], CCSR is a priority-sensitive process algebra that uses a synchronous form of concurrency, and possesses a term equivalence based on strong bisimilarity. Thus CCSR provides the ability to perform equivalence proofs by syntactic manipulation. Also, since its prioritized, strong equivalence is a congruence, it allows us to reason about a term's behavior when it is embedded in a real-time "context." CSR and CCSR share the same basic computational model, in that they are both resource-based and rely on a priority arbitration scheme to resolve resource contention.

Section 2 describes the computation model of processes and resources that underlies both CSR and CCSR. Section 3 gives an overview of CSR and shows its use in a real-time periodic producer-consumer example. Section 4 presents an overview of CCSR, and demonstrates how the example of Section 3 can be translated to the CCSR formalism. We conclude the paper in Section 5 by pointing to areas of future research.

# THE COMPUTATION MODEL

**Events.** Our basic unit of computation is the *event*, which we use to model both local resource execution as well as inter-resource synchronization. An event is executed by at most one resource and consumes exactly one time unit. This does not imply that all actions require exactly the same amount of time, rather that the event is a common infinitesimal unit, a building-block with which more complex functions are constructed. We let $\Sigma$ be the universal set of events.

**Actions.** In a system composed of multiple resources executing in parallel, the system as a whole may execute a set of events simultaneously. We call such a set of simultaneous events an *action*, which is represented by a set in $\mathcal{P}(\Sigma)$. In general, we let the letters $a$, $b$ and $c$ range over the event set $\Sigma$, and the letters $A$, $B$ and $C$ range over the action set $\mathcal{P}(\Sigma)$.

We let the Greek letters $\Delta$ and $\Gamma$ range over $\mathcal{P}(\mathcal{P}(\Sigma))$, or subsets of the action domain.

**Resources.** We let $\mathcal{R}$ represent the set of resources available to a system, and let $i$, $j$, and $k$ range over $\mathcal{R}$. Since an event is executed by at most one resource, $\Sigma$ is partitioned into mutually disjoint subsets, each of which can be considered the set of events available to a single resource. For all $i$ in $\mathcal{R}$ we denote $\Sigma_i$ as the collection of events exclusively "owned" by resource $i$:

$$\forall i \in \mathcal{R}, \forall j \in \mathcal{R} \,.\, i \neq j,\ \Sigma_i \cap \Sigma_j = \emptyset$$

Furthermore, individual resources are considered to be inherently sequential in nature. That is, a single resource is capable of synchronously executing actions that consist, *at most*, of a single event. We formalize this by defining $\mathcal{D}$, the domain of actions executable by any set of resources, as:

$$\mathcal{D} \quad = \quad \{A \in p(\Sigma) \,|\, \forall i \in \mathcal{R},\ |A \cap \Sigma_i| \leq 1\}$$

where $p(\Sigma)$ denotes the set of finite subsets of $\Sigma$. For a given action $A$, we use the notation $\mathcal{R}(A)$ to represent the resource set that executes events in $A$: $\mathcal{R}(A) = \{i \in \mathcal{R} \,|\, \Sigma_i \cap A \neq \emptyset\}$.

**Priority.** At any point in time, many events may be competing for the ability to execute on a single resource. We arbitrate such competition through the use of a priority ordering over $\Sigma$. There is a finite range of priorities at which events may execute. Letting $mp$ be the maximum possible priority, we denote $PRI = [0, \ldots, mp] \subseteq \boldsymbol{N}$ as the set of priorities available to events in the system. Thus we can linearly order the events in $\Sigma$ by a priority mapping $\pi \in \Sigma \rightarrow PRI$.

It is often necessary to consider the relative priority of *actions*. Since an action consists of multiple events from different resources, we first consider the priority of an action with respect to a *single* resource: We extend $\pi$ to singleton (and empty) sets in $\mathcal{P}(\Sigma)$ with the function $\Pi \in \mathcal{P}(\Sigma) \rightarrow PRI$ where for each $A$ in $\mathcal{P}(\Sigma)$,

$$\Pi(A) = \begin{cases} \pi(a) & \text{if } A = \{a\} \\ 0 & \text{otherwise} \end{cases}$$

We can now define the partial ordering "$\leq_p$" that reflects our notion of priority over the domain $\mathcal{D}$. For all $A, B \in \mathcal{D}$,

$$A \leq_p B \quad \text{iff} \quad \forall i \in \mathcal{R},\ \Pi(A \cap \Sigma_i) \leq \Pi(B \cap \Sigma_i)$$

# 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 some ways, it syntactically resembles the variants of real-time CSP found in [1] and [6]. However, it significantly differs by including features that take full advantage of our priority semantics. Furthermore, it has the capacity to specify many constructs commonly found in real-time systems, such as timeouts, periodic processes, and exception-handling.

## The Role of Events in CSR

Events in CSR can be used to model local computation, "input" communication and "output" communication. If an event models local computation, it can be denoted by any lower-case identifier such as "*a*", "*b*", "*do_it*", or the like. Or, if an event models input communication, it is represented by an identifier followed by a question mark, such as "*in_p*?", "*a*?", etc. If an event models output communication, it is represented by an identifier followed by an exclamation point, such as "*out*!" or "*a*!". To avoid confusion, we place the following restriction on $\Sigma$: If an identifier "*id*" is used as a local computation event, there are no events "*id*?" or "*id*!" in $\Sigma$, and *vice versa.*

In CSR, all communication between resources is strictly one-to-one, and is performed by synchronizing events. This means that there is a single resource $R_1$ that may utilize "*a*!" to denote a "write" action, and a single resource $R_2$ that may use "*a*?" to denote a "read." When both resources simultaneously agree to communicate, the two events perform a "semantic match" and the events are resolved and thus can be executed On the other hand, events that model local computation are already considered resolved, as they need no communicating partners to ensure their execution.

## The Syntax and Informal Semantics of CSR

In CSR, the system consists of a set of resources, each of which is a set of processes. A process is assigned to exactly one resource, and can engage in execution or synchronizing events. The following is a complete grammar for the CSR language:

$$
\begin{aligned}
\langle \text{system} \rangle &::= \langle \text{resource} \rangle \mid \langle \text{system} \rangle \parallel \langle \text{system} \rangle \\
\langle \text{resource} \rangle &::= \{\langle \text{process} \rangle\} \\
\langle \text{process} \rangle &::= \langle \text{proc\_id} \rangle :: \langle \text{stmt} \rangle \mid \langle \text{process} \rangle \,\&\, \langle \text{process} \rangle \\
\langle \text{stmt} \rangle &::= \textbf{wait}\ \text{t} \mid \textbf{skip} \mid a? \mid a! \mid \textbf{exec}(a, m, n) \mid \\
&\quad \langle \text{stmt} \rangle \ ; \langle \text{stmt} \rangle \mid \langle \text{guard\_s} \rangle \mid \langle \text{within\_s} \rangle \mid \\
&\quad \langle \text{interrupt\_s} \rangle \mid \langle \text{loop\_s} \rangle \mid \langle \text{every\_s} \rangle \\
\langle \text{guard\_s} \rangle &::= [\langle \text{gd} \rangle \rightarrow \langle \text{stmt} \rangle \Box \quad \ldots \Box \langle \text{gd} \rangle \rightarrow \langle \text{stmt} \rangle \\
&\qquad \triangle\ \textbf{wait}\ \text{t} \rightarrow \langle \text{stmt} \rangle] \mid \\
&\quad [\langle \text{gd} \rangle \rightarrow \langle \text{stmt} \rangle \Box \quad \ldots \Box \langle \text{gd} \rangle \rightarrow \langle \text{stmt} \rangle] \\
\langle \text{gd} \rangle &::= a \mid a? \mid a! \\
\langle \text{within\_s} \rangle &::= \textbf{within}\ \text{t}\ \textbf{do}\ \langle \text{stmt} \rangle\ \textbf{when}\ \ \text{t} \rightarrow \langle \text{stmt} \rangle\ \textbf{od} \\
\langle \text{interrupt\_s} \rangle &::= \textbf{interrupt}\ a\ \textbf{do}\ \langle \text{stmt} \rangle\ \textbf{when}\ a? \rightarrow \langle \text{stmt} \rangle\ \textbf{od} \\
\langle \text{loop\_s} \rangle &::= \textbf{loop}\ \textbf{do}\ \langle \text{stmt} \rangle\ \textbf{od} \\
\langle \text{every\_s} \rangle &::= \textbf{every}\ \text{t}\ \textbf{do}\ \langle \text{stmt} \rangle\ \textbf{od}
\end{aligned}
$$

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, $a!$. 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 [6]. 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. Note that local events are allowed as guards, in which case no delay is necessary; the priority arbitration occurs immediately. Furthermore, 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 [3], 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.

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. After all processes have been assigned to a resource, the resource is closed using the *close* operator, or "{ . }." And after a resource is closed, no other processes may be assigned to it. Only a closed resource can be combined in parallel with other closed 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. That is, we insist that communication is one-to-one *between* resources. A resource is incapable of communicating with itself since actions on a resource are purely interleaved; thus it is impossible for the read and write actions to occur simultaneously. 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, they must explicitly be modeled through guarded statements.

One final restriction is that no two resources may share a single local event. 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.

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

$$Q = P_1; (P_2 \parallel P_3); P_4$$

## Example

We now show how CSR can be used to specify a periodic producer-consumer example. In this example, there are four processes: producers P1 and P2, and consumers C1 and C2. While the producers execute on their own resources, C1 and C2 share a resource. C1 (C2) consumes input produced by P1 (P2) using the synchronizing event i1 (i2).

```
{ P1:: every 6 do
        p1;
        [i1! → exec(p3,3,3) □ wait 2 → skip ]
        od } ||
{ P2:: every 6 do
        p2;
        [ i2! → exec(p4,1,1) □ wait 4 → skip ]
        od} ||
{C1:: loop do i1?; exec(c1,2,2) od &
  C2:: loop do i2?; exec(c2,2,2) od }
```

Since P1 and P2 execute on dedicated resources, there is no contention among events; we therefore assume that $\pi(p1) = \pi(p2) = \pi(p3) = \pi(p4) = 1$ and that $\pi(11!) = \pi(i2!) = 0$, which is also the priority of waiting. For events on the shared resource, we assume the following priorities: $\pi(c1) = \pi(c2) = 3$, $\pi(i1?) = 2$, $\pi(i2?) = 1$. This priority

assignment makes $\Pi(\{i1?, i1!\}) <_p \Pi(\{i2!, i2?\})$, thus if $i1?$ and $i2?$ are simultaneously ready, $i1?$ is preferred over $i2?$. Furthermore, if $c1$ (or $c2$) is executing, it cannot be interrupted by any other events.

Informally, the system behaves as in the following diagram, where time is assumed to increase horizontally to the right:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|---|---|---|---|---|---|---|---|---|---|
| P1: | p1 | i1! | p3 | p3 | p3 | | p1 | i1! | ... |
| P2: | p2 | | | | i2! | p4 | p2 | | ... |
| C1&C2: | | i1? | c1 | c1 | i2? | c2 | c2 | i1? | ... |

At time 0, P1 and P2 can both execute their first action, p1 and p2, respectively. P1 and P2 are both ready to synchronize on i1 and i2 respectively, but since $\pi(i1) > \pi(i2)$, C1 and P1 succeed. P1 executes p3 for 3 time units, and then idles another time unit waiting to re-execute the body of **every** 6 **do**. Meanwhile, C1 executes c1 for 2 time units; C2 then synchronizes with P2 on i2 at time 4, executes c2 for 2 time units and becomes ready to re-synchronize with P1 and P2 at time 7. P2, meanwhile, executes p4 for 1 time unit and immediately starts the body of **every** 6 **do** again. P1 and P2 are then both in the same position at time 6 as they were at time 0, and the scenario repeats itself.

What we would like to be able to say about the system is that P1 and P2 never execute their **wait** statements, i.e. they never skip the execution of p3 and p4. While this can be argued informally, we would like to develop a proof system in which we can prove it formally with syntactic manipulation. As a first step, we have developed CCSR.

# A CALCULUS FOR COMMUNICATING SHARED RESOURCES

The CSR formalism adequately captures the temporal properties of prioritized resource interaction through a semantics based on linear histories. However, this semantics does not easily lend itself to an equational characterization of the CSR language. For this reason, we developed the Calculus for Communicating Shared Resources, or CCSR. Strongly influenced by SCCS [4, 5], CCSR is a process algebra that uses a synchronous form of concurrency, and possesses a term equivalence based

on strong bisimilarity [7]. CSR is syntactically a "richer" formalism than CCSR, in that it contains many real-time language constructs such as timed interrupt-handlers, temporal scopes [3], and periodic processes; however, we believe that CSR constructs can be translated into CCSR. Furthermore, CCSR provides the ability to perform equivalence proofs by syntactic manipulation.

Before describing CCSR, we introduce some notation for termination and synchronization.

**Termination.** There is one event that has special meaning: the termination event, or "$\surd$", which can be executed by every resource. We often use the following notation: for any set $A \subseteq \Sigma$, $A^{\surd}$ means $A \cup \{\surd\}$ and $A^{\not\surd}$ means $A - \surd$. Also, $\surd$ is a fixed point of all event renaming functions $\phi \in \Sigma \rightarrow \Sigma$; i.e., for all such $\phi$, $\phi(\surd) = \surd$.

**Synchronization.** The lowest form of communication is accomplished through the simultaneous execution of synchronizing events. Our model treats such synchronizing events as being statically "bound" together by the various connections between system resources. To formally treat this property we make use of *connection sets.* A connection set is a set of events that exhibits the "all or none" property of event synchronization: At time $t$, if any of the events in a given connection set wish to execute, they all must execute. A familiar example of this concept can be drawn from CSP [8, 9], where the alphabet of events is $\{c_1!, c_1?, c_2!, c_2?, c_3!, c_3?, \ldots\}$, where "$c_i$" is a channel, "$c_i!$" is interpreted as a write action, and "$c_i?$" is interpreted as a read action. When a read and a write occur simultaneously on the same channel, the communication is considered successful. The connection sets in such languages are simply $\{c_1!, c_1?\}$, $\{c_2!, c_2?\}$, etc. For the rest of this section, we assume that connections sets contain either one or two elements; one element sets model local execution, and two element sets model one-to-one communication.

Given an action $A$, we say that $A$ is *fully synchronized* iff, for every input event $a?$ in $A$, the corresponding output event, $a!$, is also in $A$ (and vice versa). Note that an action containing only local execution events is trivially fully synchronized. We define the predicate $synch(A)$ to be true iff $A$ is fully synchronized. Also, we say that $A$ is synchronized with

respect to a resource set $I \subseteq \mathcal{R}$ iff for every input-output pair $\{a!, a?\}$ in $\bigcup_{j \in I} \Sigma_j$, if the input event $a?$ is in $A$, the corresponding output event, $a!$, is also in $A$ (and vice versa). We define the predicate $sync_{(I)}(A)$ to be true iff $A$ is fully synchronized with respect to the resource set $I$.

It is often convenient to be able to decompose an action $A \in \mathcal{P}(\Sigma)$ into two parts: that which is fully synchronized, and that which is not. To do this, we make use of the following two definitions:

$$\begin{aligned} res(A) &= \bigcup\{B \subseteq A \mid sync(B)\} \\ unres(A) &= A - res(A) \end{aligned}$$

**Priority-Canonical Events.** Priority and resource mapping naturally partition $\Sigma$ into equivalence classes. That is, for events $a, b \in \Sigma$, $a$ is in the class $[b]$ if and only if for some $i \in \mathcal{R}$, $a, b \in \Sigma_i$ and $\pi(a) = \pi(b)$. In CCSR, we use the symbol "$\tau_i^n$" to represent a "canonical" event from each class, where $\tau_i^n$ is mapped to resource $i$ and has priority $n$. Further, for every $i \in \mathcal{R}$, there is a $\tau_i^0$ in $\Sigma_i$: every resource has the capacity to execute at the lowest priority level.

There is a unique renaming function, $\phi_\pi$, such that if $a \in [\tau_i^n]$, then $\phi_\pi(a) = \tau_i^n$. It follows that the $\tau_i^n$ are fixed-points of priority renaming; that is, $\phi_\pi(\tau_i^n) = \tau_i^n$. All such "canonical" events are local with respect to their own resources; that is, they belong to their own connection sets: For each $a \in \Sigma$ there is a connection set $j \in \mathcal{C}$ such that $\{\phi_\pi(a)\} = C_j$.

## The CCSR Language

The syntax of CCSR resembles, in some respects, that of SCCS [4, 5]. Let $\mathcal{E}$ represent the domain of terms, and let $E$, $F$, $G$ and $H$ range over $\mathcal{E}$. Additionally we assume an infinite set of free term variables, $FV$, with $X$ ranging over $FV$ and $free(E)$ representing the set of free variables in the term $E$. Let $\mathcal{P}$ represent the domain of closed terms, which we call *agents* or alternatively, *processes*, and let $P$, $Q$, $R$ and $S$ range over $\mathcal{P}$. The following grammar defines the terms of CCSR:

$$\begin{aligned} E \;:=\; & NIL \mid A : E \mid E + E \mid E_I \|_J E \mid E \,\Delta_t^B\, (E, E, E) \mid [E]_I \mid E\backslash A \mid \\ & fix(X.E) \mid X \end{aligned}$$

While we give a semantics for these terms in subsequent sections, we briefly present some motivation for them here. The term $NIL$ corresponds to **0** in SCCS – it can execute no action whatsoever. The Action

operator, "$A : E$", has the following behavior. At the first time unit, the action $A$ is executed, proceeded by the term $E$. The Choice operator represents selection – either of the terms can be chosen to execute, subject to the constraints of the environment. For example, the term $(A : E) + (B : F)$ may execute $A$ and proceed to $E$, or it may execute $B$ and proceed to $F$.

The Parallel operator $E_I \| _J F$ has two functions. It defines the resources that can be used by the two terms, and also forces synchronization between them. Here, $I \subseteq \mathcal{R}$ is a set of the resources allotted to $E$, and $J \subseteq \mathcal{R}$ is a set of the resource allotted to $F$. In the case where $I \cap J \neq \emptyset$, $E$ and $F$ may be able to share certain resources. But as we have stated, such resource-sharing must be interleaved.

The Scope construct $E \Delta_t^B (F, G, H)$ binds the term $E$ by a temporal scope [3], and it incorporates both the features of timeouts and interrupts. We call $t$ the *time bound* and $B$ the *termination control*, where $t \in \mathbb{N}^+ \cup \{\infty\}$ (i.e., $t$ is either a positive integer or infinity), and $B = \{\surd\}$ or $B = \emptyset$.

While $E$ is executing we say that the scope is *active*. The scope can be exited in a number of ways, depending on the values of $E$, $H$, $t$ and $B$. If $E$ successfully terminates within time $t$ by executing "$\surd$", then $F$ is initiated. Here, if $B = \{\surd\}$, the transition from $E$ to $F$ will retain its ability to signal termination, while if $B = \emptyset$, the entire construct will terminate only when $F$ does.

There are two other ways in which the scope may be exited. If $E$ fails to terminate within $t$ units, the "exception-handler" $G$ is executed. Lastly, at any time throughout the execution of $E$, it may be interrupted by $H$, and the scope is then departed.

As an example of the Scope operator, consider the following specification: "Execute $P$ for a maximum of 100 time units. If $P$ successfully terminates within that time, then terminate the system. However, if $P$ fails to finish within 100 time units, at time 101 start executing $R$. At any time during the execution of $P$, allow interruption by an action $\{a?\}$ which will halt $P$, and initiate the interrupt-handler $S$." This system may be realized by the following term: $P \Delta_{100}^{\{\surd\}} (NIL, R, \{a?\} : S)$.

Now consider this specification: "Execute $P$ for a maximum of 100 time units. If $P$ successfully terminates within that time, "cancel" the termination and proceed to $Q$. If $P$ fails to finish within 100 time units,

at time 101 start executing $R$." This specification yields the following term: $P \Delta^{\emptyset}_{100} (Q, R, NIL)$.

We note that sequential composition may be realized by using the Scope operator. To sequentially compose $E$ and $F$, we may use this term: $E \Delta^{\emptyset}_{\infty} (F, NIL, NIL)$.

The Close operator, $[E]_I$, denotes that the term $E$ occupies *exactly* the resources represented in the index $I$. In addition, Close produces a term that totally utilizes the resources in $I$; that is, it prohibits further sharing of those resources.

The Hiding operator $E\backslash A$ masks events of action $A$ in $E$ up to their resource usage and priority, in that while the events themselves are hidden, their priorities are still observable. The term $fix(X.E)$ denotes recursion, allowing the specification of infinite behaviors.

## An Operational Semantics

In this section we present an operational semantics for closed terms, in the style of [10]. We do this in two steps. First, we define a labeled transition system $\langle \mathcal{E}, \rightarrow, \mathcal{D} \rangle$, which is a relation $\rightarrow \subseteq \mathcal{E} \times \mathcal{D} \times \mathcal{E}$. We denote each member $(E, A, F)$ of "$\rightarrow$" as "$E \xrightarrow{A} F$". We call this transition system *unconstrained*, in that no priority arbitration is made between actions. Thus, if $E \xrightarrow{A} F$ is in "$\rightarrow$", it means that in a system without preemption constraints, a term $E$ may execute $A$ and proceed to $F$. After presenting "$\rightarrow$", we use it to define a prioritized transition system $\langle \mathcal{E}, \rightarrow_\pi, \mathcal{D} \rangle$, which is sensitive to preemption. This two-phased approach greatly simplifies the definition of "$\rightarrow_\pi$"; similar tactics have been used by [11] in their treatment of CCS priority, and by [1] in their semantics for maximum parallelism.

Throughout, we use the following notation. For a given set of resources $I \subseteq \mathcal{R}$, we let $\Sigma_I$ represent the set $\bigcup_{i \in I} \Sigma_i$. Also, $A * B = A^{\surd} \cup B^{\surd} \cup (A \cap B)$; that is, the termination event "$\surd$" is an element of $A * B$ if and only if it is in both $A$ and $B$.

Tables 1 and 2 present the unconstrained transition system, "$\rightarrow$". The rules for Action, Choice and Recursion are quite straightforward. The other rules, however, require some additional explanation.

**Parallel.** The four side conditions define both the resource mapping and synchronization constraints imposed on terms that operate in a con-

**Action :** $A : E \xrightarrow{A} E$

**ChoiceL :** $\dfrac{E \xrightarrow{A} E'}{E + F \xrightarrow{A} E'}$ **ChoiceR :** $\dfrac{F \xrightarrow{A} F'}{E + F \xrightarrow{A} F'}$

**Parallel :** $\dfrac{P_1 \xrightarrow{A_1} P_1',\ P_2 \xrightarrow{A_2} P_2'}{P_1 \,{}_I\|_J\, P_2 \xrightarrow{A_1 * A_2} P_1' \,{}_I\|_J\, P_2'} \begin{pmatrix} A_1 \subseteq \Sigma_I^{\surd},\ A_2 \subseteq \Sigma_J^{\surd}, \\ \mathcal{R}(A) \cap \mathcal{R}(B) = \emptyset, \\ sync_{(I \cup J)}(A_1 * A_2) \end{pmatrix}$

**ScopeC :** $\dfrac{E \xrightarrow{A} E'}{E \,\triangle_t^B\, (F, G, H) \xrightarrow{A} E' \,\triangle_{t-1}^B\, (E, F, G)} \quad (t > 1,\ \surd \notin A)$

**ScopeE :** $\dfrac{E \xrightarrow{A} E'}{E \,\triangle_t^B\, (F, G, H) \xrightarrow{A * B} F} \quad (t \geq 1,\ \surd \in A)$

**ScopeT :** $\dfrac{E \xrightarrow{A} E'}{E \,\triangle_t^B\, (F, G, H) \xrightarrow{A} G} \quad (t = 1,\ \surd \notin A)$

**ScopeI :** $\dfrac{H \xrightarrow{A} H'}{E \,\triangle_t^B\, (F, G, H) \xrightarrow{A} H'} \quad (t \geq 1)$

Table 1: Unconstrained Transition System

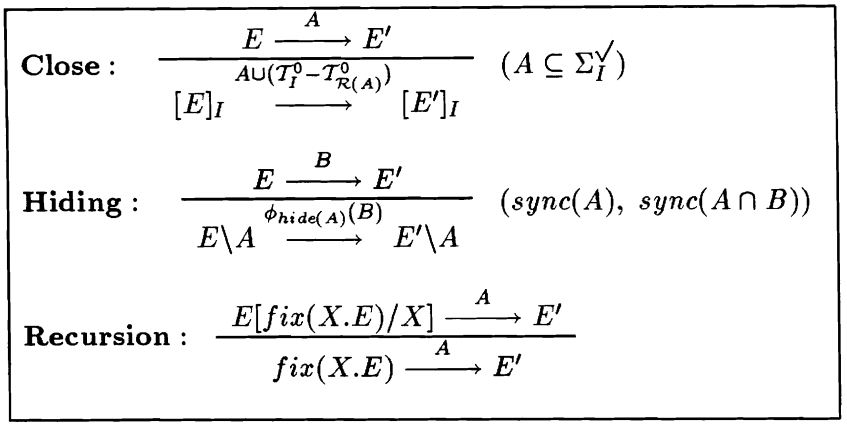


Table 2: Unconstrained Transition System, cont.

current fashion. The first two conditions define the resources on which the terms $E_1$ and $E_2$ may execute. That is, $A_1$ must be hosted on the resources denoted by $I$, while $A_2$ must be hosted on the resources denoted by $J$. Moreover, the third condition stipulates that single resources may not execute more than one event at a time.

The final side condition, "$sync_{(I \cup J)}(A_1 * A_2)$", defines our notion of inter-resource synchronization. Assume that $E_1$ can execute an action $A_1 \subseteq \Sigma_I^{\surd}$, and that $E_2$ can execute an action $A_2 \subseteq \Sigma_J^{\surd}$. Then $A_1$ and $A_2$ may execute simultaneously if and only if they are connected in the following sense: If any event in $a \in A_1$ shares a connection set with some event $b \in \Sigma_J$, then $b$ *must* appear in $A_2$, and *vice versa*. This synchronization constraint is a generalized version of that found in CSP.

**Scope.** There are four rules for the Scope operator, corresponding to the four actions that may be taken while a term $E$ is bound by a temporal scope. Assume that $E \xrightarrow{A} E'$ with $\surd \notin A$, and that $t > 1$. In such a situation, the ScopeC law is used to keep the temporal scope active; i.e., $E'$ is bound by the scope with its time limit decremented to $t - 1$. On the other hand if $\surd \in A$ and $t \geq 1$, ScopeE is used. In this case the scope is departed by executing $A * B$, at which time $F$ is initiated. By the definition of "$*$", if $B = \emptyset$, then $A * B = A - \{\surd\}$. That is, while

$E$ itself may terminate by executing $A$, the entire term will terminate when (or if) $F$ does. But if $B = \{\surd\}$, then $A * B = A$. This means that the entire term may terminate by executing $A$.

Now assume that $E \xrightarrow{A} E'$ such that $\surd \notin A$, and that $t = 1$. Thus implies that the ScopeT rule must be used ("T" is for timeout). Here, the scope has "timed out", and thus, first $A$ is executed, followed by the exception-handler $G$. Finally, the ScopeI rule shows that the term $H$ may interrupt at any time while the temporal scope is active.

**Close.** The Close operator assigns terms to occupy *exactly* the resource set denoted by the index $I$. First, the action $A$ may not utilize *more* than the resources in $I$; otherwise it is not admitted by the transition system. On the other hand, if the events in $A$ utilize less than the set $I$, the action is augmented with the "idle" events from each of the unused resources. For example, assume $E$ executes an action $A$, and that there is some $i \in I$ such that $i \notin \mathcal{R}(A)$. In $[E]_I$, this gap is filled by including $\tau_0^i$ in $A$. Here we use the notation $\mathcal{T}_J^0$ to represent *all* of the 0-priority idle events from the resource set $J$:

$$\mathcal{T}_J^0 = \{\tau_j^0 \mid j \in J\}$$

**Hiding.** Assume that $E$ executes an action $B$, and that $sync(A)$ and $sync(A \cap B)$ both hold. Then using the Hiding rule, $E\backslash A$ executes an action that reduces the events in $A \cap B$ to their "canonical" priority representation, as described in Section . If $A \subseteq \Sigma$, we construct the function $\phi_{hide(A)}$ as follows. For all $a$ in $\Sigma$,

$$\phi_{hide(A)}(a) = \begin{cases} \phi_\pi(a) & \text{if } a \in A \\ a & \text{otherwise} \end{cases}$$

Thus, $\phi_{hide(A)}(B) = (B - A) \cup \phi_\pi(B \cap A)$; i.e., all of the events in $B \cap A$ are mapped to their corresponding "canonical" events.

The reason for this nonstandard hiding construction should be clear when viewed from the perspective of resource usage. As an example, let $a, b \in \Sigma_i$, with $\{a\}$ and $\{b\}$ as connection sets; i.e., both events are completely local to resource $i$. Now let $E = \{a\} : NIL$ and $F = \{b\} : NIL$. In a more "standard" definition of hiding, $E\backslash\{a\} \xrightarrow{\emptyset} NIL\backslash\{a\}$. In other words, all "$a$" is completely abstracted from the system behavior. But in this definition, we find that $(E\backslash\{a\})_{\{i\}} \|_{\{i\}} F \xrightarrow{\{b\}}$

$(NIL\backslash\{a\})_{\{i\}} \|_{\{i\}} NIL$. This would violate the resource-based execution model, in that two events from resource $i$ would execute simultaneously.

**Proposition 1** *All terms in $\mathcal{E}$ are well-defined, in that if $E \in \mathcal{E}$ and $E \xrightarrow{A} E'$, then $A \in \mathcal{D}$.*
The proof follows directly from the definition of the operators. □

We define a prioritized transition system based on a preemption measure, $\prec$, as follows.

**Definition 1** *For all $A \in \mathcal{D}$, $B \in \mathcal{D}$, $A \preceq B$ if and only if*

$$\mathcal{R}(A) = \mathcal{R}(B) \wedge unres(A) = unres(B) \wedge res(A) \leq_p res(B)$$

*The relation "$\preceq$" defines a partial order over $\mathcal{D}$ and thus, we say $A \prec B$ if $A \preceq B$ and $B \not\preceq A$, i.e., $\mathcal{R}(A) = \mathcal{R}(B) \wedge unres(A) = unres(B) \wedge res(A) <_p res(B)$.* □

The prioritized transition system, $\langle \mathcal{E}, \rightarrow_\pi, \mathcal{D} \rangle$, is given by:

**Definition 2** *The labeled transition system $\langle \mathcal{E}, \rightarrow_\pi, \mathcal{D} \rangle$ is a relation $\rightarrow_\pi \in \mathcal{E} \times \mathcal{D} \times \mathcal{E}$ and is defined as follows: $(P, A, P') \in \rightarrow_\pi$ (or $P \xrightarrow{A}_\pi P'$) if:*

1. *$P \xrightarrow{A} P'$, and*
2. *For all $A' \in \mathcal{D}$, $P'' \in \mathcal{E}$ such that $P \xrightarrow{A'} P''$, $A \not\prec A'$.* □

## Bisimulation and Priority Equivalence

In our semantics, equivalence between agents is based on the concept of *strong bisimulation* [7], which is formally defined as follows:

**Definition 3** *For a given transition system $\langle \mathcal{E}, \rightsquigarrow, \mathcal{D} \rangle$, the symmetric relation $r \subseteq (\mathcal{E}, \mathcal{E})$ is a* strong bisimulation *if, for $(P, Q) \in r$ and $A \in \mathcal{D}$,*

1. *if $P \overset{A}{\rightsquigarrow} P'$ then, for some $Q'$, $Q \overset{A}{\rightsquigarrow} Q'$ and $(P', Q') \in r$, and*
2. *if $Q \overset{A}{\rightsquigarrow} Q'$ then, for some $P'$, $P \overset{A}{\rightsquigarrow} P'$ and $(P', Q') \in r$.* □

We let "$\sim_\pi$" denote the largest strong bisimulation over the transition system $\langle \mathcal{E}, \rightarrow_\pi, \mathcal{D} \rangle$, and we call it *prioritized strong equivalence*. Relying on the well-known theory in[4, 5], "$\sim_\pi$" exists. We have shown that "$\sim_\pi$" forms a congruence over the CCSR operators. This property is nice since it allows us to develop a compositional proof system. Indeed, we found a set of equational laws with respect to $\sim_\pi$, which forms a proof system for CCSR terms (see Tables 3 and 4).

## An Example

To show the relationship between CSR and CCSR, we will translate the producer-consumer example of the previous section into CCSR. First, we introduce some notation that facilitates a concise specification. For a term $P$ and a nonnegative integer $t$, let "$\delta_t P$" be the term that *may* delay the execution of $P$ for $t$ time units, and if $P$ is not executed by then it will idle forever. That is:

$$\delta_t P = \begin{cases} \emptyset^\infty & \text{if } t = 0 \\ P \ + \ (\emptyset : \delta_{t-1} P) & \text{otherwise} \end{cases}$$

Furthermore, let $A^i$ be shorthand for the Action operator with $A$ repeated $i$ times. For example, $A^2 : P$ is shorthand for $A : A : P$.

To translate P1, we use the $\delta_2$ operator to indicate how long P1 can delay before i1! is accepted. We also pad P1's execution with $\emptyset$ to ensure that its normal execution will not finish before 6 times units after the beginning of each period. This ensures that P1 times-out at the end of each period, and that P1 repeats, i.e. starts a new period. P2 is similarly defined.

$$P1 = (\{p1\} : \delta_2\{i1!\} : (\{p3\})^3 : \emptyset^\infty) \, \Delta_6^\emptyset \, (NIL, P1, NIL)$$
$$P2 = (\{p2\} : \delta_4\{i2!\} : \{p4\} : \emptyset^\infty) \, \Delta_6^\emptyset \, (NIL, P2, NIL)$$

The consumer processes are straightforward: Consumer C1 waits for input from P1 forever, executes c1 for two time units, and repeats. C2 is similarly defined.

$$C1 = \ \delta_\infty\{i1?\} : (\{c1\})^2 : C1$$
$$C2 = \ \delta_\infty\{i2?\} : (\{c2\})^2 : C2$$

| | |
|---|---|
| Choice(1) | $E + NIL = E$ |
| Choice(2) | $E + E = E$ |
| Choice(3) | $E + F = F + E$ |
| Choice(4) | $(E + F) + G = E + (F + G)$ |
| Choice(5) | $(A : E) + (B : F) = B : F \quad \text{if } A \prec B$ |
| Par(1) | $E_I \|\|_J NIL = NIL$ |
| Par(2) | $E_I \|\|_J F = F_J \|\|_I E$ |
| Par(3) | $(E_I \|\|_J F)_{(I \cup J)} \|\|_K G = E_I \|\|_{(J \cup K)} (F_J \|\|_K G) \quad \text{if } I \cap J \cap K = \emptyset$ |
| Par(4) | $E_I \|\|_J (F + G) = (E_I \|\|_J F) + (E_I \|\|_J G)$ |
| Par(5) | $(A : E)_I \|\|_J (B : F) =$ $\begin{cases} (A * B) : (E_I \|\|_J F) & \text{if } A \subseteq \Sigma_I^{\surd},\ B \subseteq \Sigma_J^{\surd}, \\ & \mathcal{R}(A) \cap \mathcal{R}(B) = \emptyset,\ sync_{(I \cup J)}(A * B) \\ NIL & \text{otherwise} \end{cases}$ |
| Scope(1) | $NIL \, \Delta_t^B \, (F, G, H) = H$ |
| Scope(2) | $(E_1 + E_2) \, \Delta_t^B \, (F, G, H) = (E_1 \, \Delta_t^B \, (F, G, H)) \; + \; (E_2 \, \Delta_t^B \, (F, G, H))$ |
| Scope(3) | $(A : E) \, \Delta_t^B \, (F, G, H) = \begin{cases} (A * B : F) + H & \text{if } \surd \in A \\ (A : (E \, \Delta_{t-1}^B \, (F, G, H))) + H & \text{if } \surd \notin A \text{ and } t > 1 \\ (A : G) + H & \text{otherwise} \end{cases}$ |
| Close(1) | $[NIL]_I = NIL$ |
| Close(2) | $[E + F]_I = [E]_I + [F]_I$ |
| Close(3) | $[A : E]_I = \begin{cases} (A \cup (\mathcal{T}_I^0 - \mathcal{T}_{\mathcal{R}(A)}^0)) : [E]_I & \text{if } A \subseteq \Sigma_I^{\surd} \\ NIL & \text{otherwise} \end{cases}$ |

Table 3: The Axiom System, $\mathcal{A}$

| | |
|---|---|
| Close(4) | $[[E]_I]_J = \begin{cases} [E]_J & \text{if } I \subseteq J \\ NIL & \text{otherwise} \end{cases}$ |
| Hide(1) | $NIL\backslash B = NIL$ |
| Hide(2) | $(E+F)\backslash B = E\backslash B + F\backslash B$ |
| Hide(3) | $(A:E)\backslash B = \begin{cases} \phi_{hide(B)}(A) : (E\backslash B) & \text{if } sync(A \cap B) \\ NIL & \text{otherwise} \end{cases}$ |

Table 4: The Axiom System, $\mathcal{A}$, cont.

As for the connection sets, let $C_1 = \{i1!, i1?\}$, $C_2 = \{i2!, i2?\}$, $C_3 = \{p1\}$, $C_4 = \{p2\}$, $C_5 = \{p3\}$, $C_6 = \{p4\}$, $C_7 = \{c1\}$, and $C_8 = \{c2\}$. That is, resources 1 and 3 are connected by $C_1$, resources 2 and 3 are connected by $C_2$, while all other events are local to the resources that own them. The priorities are the same as before.

The entire system is described as follows:

$$System = [\,(P1_{\{1\}} \| _{\{2\}} P2)_{\{1,2\}} \| _{\{3\}} (C1_{\{3\}} \| _{\{3\}} C2)\,]_{\{1,2,3\}}$$

By definition of $\delta_t$, $P1$ and $P2$ are equivalent ($\sim_\pi$) to $P1'$ and $P2'$ respectively:

$$\begin{aligned} P1' = (\{p1\} : (\{i1!\} : \{p3\}^3 : \emptyset^\infty \\ + \emptyset : (\{i1!\} : \{p3\}^3 : \emptyset^\infty \\ + \emptyset^\infty))) \, \Delta_6^\emptyset \, (NIL, P1', NIL) \end{aligned}$$

$$\begin{aligned} P2' = (\{p2\} : (\{i2!\} : \{p4\} : \emptyset^\infty \\ + \emptyset : (\{i2!\} : \{p4\} : \emptyset^\infty \\ + \emptyset : (\{i2!\} : \{p4\} : \emptyset^\infty \\ + \emptyset : (\{i2!\} : \{p4\} : \emptyset^\infty \\ + \emptyset^\infty))))) \, \Delta_6^\emptyset \, (NIL, P1', NIL) \end{aligned}$$

Combining $P1'$ and $P2'$ using repeated applications Par(5) and Choice(5), we get:

$$(P1'_{\{1\}}\|_{\{2\}} P2') \sim_\pi$$
$$\text{fix } X.(\{p1,p2\} : (\{i1!,i2!\} : \{p3,p4\} : \{p3\} : \{p3\} : \emptyset^\infty$$
$$+\{i1!\} : (\ \{p3,i2!\} : \{p3,p4\} : \{p3\} : \emptyset^\infty$$
$$+\{p3\} : (\{p3,i2!\} : \{p3,p4\} : \emptyset^\infty$$
$$+\{p3\} : (\ \{p3,i2!\} : \{p4\} : \emptyset^\infty$$
$$+\{p3\} : \emptyset^\infty)))$$
$$+\{i2!\} : (\{i1!,p4\} : \{p3\} : \{p3\} : \{p3\} : \emptyset^\infty$$
$$+\{p4\} : (\{i1!\} : \{p3\} : \{p3\} : \{p3\} : \emptyset^\infty$$
$$+\emptyset^\infty))$$
$$+\emptyset : (...))\ \Delta_6^\emptyset\ (NIL, X, NIL))$$

Combining $C1$ and $C2$ using repeated applications of Par(5), and noting that the side condition $\mathcal{R}(A) \cap \mathcal{R}(B) = \emptyset$ prevents two events from the same resource appearing in the same action, we get:

$$(C1_{\{3\}}\|_{\{3\}} C2) \sim_\pi$$
$$\text{fix } X.(\emptyset : X + \{i1?\} : (\{c1\})^2 : X + \{i2?\} : (\{c2\})^2 : X)$$

Using the rules for recursion, Par(5), and Choice(5) (noting that the side condition $sync_{(I\cup J)}(A * B)$ forces synchronizing events to occur in the same action), we can formally show that our system is equivalent to the following agent. Note that $P1$ and $P2$ execute p3 and p4 each period.

$$System = [\{p1,p2\} : \mathbf{fix}X.(\ \{i1!,i1?\} : \{p3,c1\} : \{p3,c1\} : \{p3,i2!,i2?\} :$$
$$\{p4,c2\} : \{p1,p2,c2\} : X)]_{\{1,2,3\}}$$

# SUMMARY AND FUTURE WORK

We developed a resource-based model of computation, in which multiple resources execute synchronously, while processes assigned to the same resource are interleaved according to their priorities. Using this model, we specify the behavior of distributed real-time systems using Communicating Shared Resources. Although CSR is good for specifying distributed real-time systems, it is too complex to be treated as a process algebra, and is therefore not amenable for developing a proof system. We therefore developed a calculus for CSR, CCSR, based on the same resource based computation model. The CCSR syntax includes primitive

constructs to express essential real-time functionality, among which are timeouts, interrupts, periodic behaviors and exceptions. Further, there is a single parallel operator that can be used to express both interleaving at the resource level, and "true" concurrency at the system level.

CCSR's proof system derives from a term equivalence based on strong bisimulation, which incorporates a notion of preemption based on priority, synchronization and resource utilization. This prioritized equivalence is also a congruence, which leads to the compositionality of our proof system. Thus we can prove correctness for a real-time system by modularly reasoning about its subsystems, the usefulness of which was shown in [12].

Proving equivalence for two CCSR terms can be a long and laborious process, as it often involves the manipulation of very complex expressions. Thus, we plan to mechanize the procedure using the HOL theorem prover [13], which can assist a human both in structuring these proofs, as well as in verifying their correctness. We are also constructing a reachability analyzer for CCSR terms. This tool will use the transition system to construct a reachability graph for a given program. The generated graph can then provide the foundation for model checking, and can be applied toward decision procedures that distinguish certain properties, such as deadlock, liveness, or the equivalence of two different programs. Finally, we are automating a translation procedure from the CSR programming language [14] to the CCSR process algebra. This translator will preserve the semantic characteristics of CSR programs, and thus, it will enable us to reason about them using the CCSR proof system.

## References

[1] C. Huizing, R. Gerth, and W. de Roever, "Full Abstraction of a Denotational Semantics for Real-time Concurrency," in *Proc. $14^{th}$ ACM Symposium on Principles of Programming Languages*, pp. 223–237, 1987.

[2] R. Koymans, R. Shyamasundar, W. de Roever, R. Gerth, and S. Arun-Kumar, "Compositional Semantics for Real-Time Distributed Computing," *Information and Computation*, vol. 70, 1988.

[3] I. Lee and V. Gehlot, "Language Constructs for Distributed Real-Time Programming," in *Proc. IEEE Real-Time Systems Symposium*, 1985.

[4] R. Milner, "Calculi for synchrony and asynchrony," *Theoretical Computer Science*, vol. 25, pp. 267–310, 1983.

[5] R. Milner, *Communication and Concurrency.* Prentice-Hall, 1989.

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

[7] D. Park, "Concurrency and Automata on Infinite Sequences," in *LNCS 104*, 1981.

[8] C. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[9] C. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, pp. 666–676, August 1978.

[10] G. Plotkin, "A structural approach to operational semantics," Tech. Rep. DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

[11] R. Cleaveland and M. Hennessy, "Priorities in Process Algebras," *Information and Computation*, vol. 87, pp. 58–77, 1990.

[12] R. Gerber and I. Lee, "A Proof System for Communicating Shared Resource," in *Proc. 11th IEEE Real-Time Systems Symposium*, 1990.

[13] M. Gordon, "HOL – A Proof Generating System for Higher-Order Logic," in *Proc. Hardware Verification Workshop, Calgary, Canada*, 1987.

[14] R. Gerber and I. Lee, "Communicating Shared Resources: A Model for Distributed Real-Time Systems," in *Proc. 10th IEEE Real-Time Systems Symposium*, 1989.