A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems

MS-CIS-93-08
LOGIC & COMPUTATION 56
DISTRIBUTED SYSTEMS LAB 14

Insup Lee
Patrice Brémond-Grégoire
Richard Gerber

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

January 1993

A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems *

Insup Lee, Patrice Brémond-Grégoire
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

Richard Gerber
Department of Computer Science
University of Maryland
College Park, MD 20742

May 1, 1993

Abstract

Recently, significant progress has been made in the development of timed process algebras for the specification and analysis of real-time systems. This paper describes a timed process algebra called ACSR, which supports synchronous timed actions and asynchronous instantaneous events. Timed actions are used to represent the usage of resources and to model the passage of time. Events are used to capture synchronization between processes. To be able to specify real systems accurately, ACSR supports a notion of priority that can be used to arbitrate among timed actions competing for the use of resources and among events that are ready for synchronization. The paper also includes a brief overview of other timed process algebras and discusses similarities and differences between them and ACSR.

*This research was supported in part by ONR N00014-89-J-1131, DARPA/NSF CCR90-14621 and NSF CCR 92-09333.

1 Introduction

Reliability in real-time systems can be improved through the use of formal methods for the specification and analysis of real-time systems. Formal methods treat system components as mathematical objects and provide mathematical models to describe and predict the observable properties and behaviors of these objects. There are several advantages to using formal methods for the specification and analysis of real-time systems. They are, firstly, the early discovery of ambiguities, inconsistencies and incompleteness in informal requirements; secondly, the automatic or machine-assisted analysis of the correctness of specifications with respect to requirements; and thirdly, the evaluation of design alternatives without expensive prototyping.

Recently, there has been significant progress in the development of real-time formal methods [32, 21, 18, 15, 25, 19, 37, 36, 10, 24, 26, 1]. Much of this work falls into the traditional categories of untimed systems such as temporal logics, assertional methods, net-based models, automata theory and process algebras. In this paper, we provide an overview of the *Algebra of Communicating Shared Resources* (ACSR), which is a real-time process algebra that we have developed.

Process algebras, such as CCS [29], CSP [17], Acceptance Trees [13] and ACP [4], have been developed to describe and analyze communicating, concurrently executing systems. They are based on the premises that the two most essential notions in understanding complex dynamic systems are concurrency and communication [29]. The most salient aspect of process algebras is that they support the *modular* specification and verification of a system. This is due to the algebraic laws that form a compositional proof system, and thus, it is possible to verify the whole system by reasoning about its parts. Process algebras without the notion of time are being used widely in specifying and verifying concurrent systems. To expand their usefulness to real-time systems, several real-time process algebras have been developed by adding the notion of time and including a set of timing operators to process algebras.

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. Most current real-time process algebras adequately capture delays due to process synchronization; however, they abstract out resource-specific details by assuming idealistic operating environments. On the other hand, scheduling and resource allocation algorithms used for real-time systems ignore the effect of process synchronization except for simple precedence relations between processes. Our algebra provides a formal framework that combines the areas of process algebra and real-time scheduling, and thus, can help us to reason about systems that are sensitive to deadlines, process interaction and resource availability.

The computation model of ACSR is based on the view that a real-time system consists of a set of communicating processes that use shared resources for execution and synchronize with one another. The use of shared resources is represented by timed actions and synchronization is supported by instantaneous events. The execution of a timed action is assumed to take one time unit and to consume a set of resources during that time. Idling of a process is treated as a special timed action that consumes no resources. The execution of a timed action is subject to the availability of the resources it uses. The contention for resources is arbitrated according to the priorities of competing actions. To ensure the uniform progression of time, processes execute timed actions synchronously. Unlike a timed action, the execution of an event is instantaneous and never consumes any resource. Processes execute events asynchronously except when two processes synchronize through matching events. Priorities are used to arbitrate the choice when several events are possible at the same time.

The rest of the paper is organized as follows. Section 2 provides a general overview of real-time extensions to process algebras. In Section 3, we briefly introduce the computation model. In Section 4, we present the syntax of the algebra and describe operational semantics and examples. Section 5 defines the notion of equivalence and describes a set of equational laws that can be used to show the equivalence of two terms through syntactic manipulation. The dining philosophers example is used to illustrate the use of the equational laws in proving that a specification is correct. Section 6 relates the algebra described in this paper with other real-time process algebras. In particular, we compare the main features that distinguish these algebras, including time models, timed operators and priorities. We conclude in Section 7 by discussing possible extensions to the process algebraic approach.

2 Overview of Real-Time Process Algebras

Following the pioneering work on CCS [27], CSP [16] and ACP [4], process algebras have been extensively used as a vehicle for the better understanding of concurrency and communication. In general, a process algebra consists of 1) a set of operators and syntactic rules for constructing processes; 2) a semantic mapping which assigns meaning or interpretation to processes; 3) a notion of equivalence or partial order between processes; and 4) a set of algebraic laws that allows syntactic manipulation of processes. Recently, several extensions have been made to process algebras to incorporate the notion of time. We now discuss several design issues in developing a real-time process algebra and explain our choices.

Time. The important aspect of a real-time process algebra is the ability to capture the time at which an event occurs. The domain of time is totally ordered, i.e., for every two elements that are not equal, one is considered to be earlier than the other. The domain of time can be either discrete or dense. A time domain is discrete if each element in the domain has a unique successor; that is, events can occur only at fixed time intervals. A time domain is dense if there is an element between any two elements in the domain; that is, an event can happen at any arbitrary moment in time. Although algebras with the dense time domain are more expressive than those with the discrete time domain, for most algebras the complete axiomatization is possible only for a discrete time domain. Thus, most real-time process algebras, including ACSR, are based on discrete time.

Concurrency Semantics. With the addition of time, the behavior of a process is a sequence of event-time pairs, $\langle (a_1, t_1), (a_2, t_2) \ldots (a_n, t_n) \rangle$, where time denotes the occurrence time of the event. There are two approaches to the treatment of events with the same occurrence time. One is to view them occurring asynchronously, that is, one after another. This approach, for example, treats a timed behavior $\langle (a, 10), (b, 10) \rangle$ differently from $\langle (b, 10), (a, 10) \rangle$. This approach can be justified as follows: although the two events are observed to occur at the same time due to the use of discrete time, there may be a causal dependency between them; for example, the action a might have caused the action b to occur. The other approach is to view them occurring synchronously, that is, both of them are occurring at the same time. Thus, the above two timed behaviors are considered equal and can be represented uniquely by $\langle (\{a, b\}, 10) \rangle$.

Operators. A term of a process algebra represents a process that is transformed into another process after executing some action. Process algebras include a set of operators, which are used to construct a complex process term from simpler components. All process algebras provide operators that are functionally similar to the following operators: prefix for sequencing of actions; choice or plus for choosing between alternatives; parallel for composing two processes to run in parallel; restriction or hiding for abstracting the details or names of actions; relabeling or renaming for changing the names of actions; and recursion for describing processes with infinite behaviors. In addition, real-time process algebras support a variety of operators that deal with time. They are basically to *delay* execution for t time units, to *timeout* while waiting for some actions to occur, and to *bound* the time it takes to execute a sequence of actions. However, as we will discuss in Section 6, each process algebra includes a different set of timed operators to support the above three capabilities.

Communication. Communication is essential in achieving the cooperation of concurrent processes. The common communication primitives are either 2-way synchronous communication or n-way synchronous communication. The 2-way synchronous communication allows only two processes to synchronize at a time, and thus, communicating actions are restricted to an action (say a) and its inverse action ($\bar{a}$). To prevent further synchronization with an already established communication, the effect of communication between two actions, a and $\bar{a}$, is to convert the simultaneous occurrence of the two actions into the internal τ action. For example, in

$$a.P \| \bar{a}.Q \| a.R = (\tau.(P \| Q) \| a.R) + (a.P \| \tau.(Q \| R)),$$

the left-hand-side process is equal to the right-hand-side process, which lets the processes $a.P$ and $\bar{a}.Q$ synchronize, or let the processes $\bar{a}.Q$ and $a.R$ synchronize, but not both at the same time.

N-way communication allows more than two processes to participate in synchronization. This means that actions and their inverses are not converted into a special action, such as τ, since the possibility of further synchronization has to be allowed. For example, in

$$a.P \| a.Q \| a.R = a.(P \| Q \| R),$$

the left-hand-side process is equal to the right-hand-side process, in which all the three processes are synchronized on the action a.

Abstraction. When dealing with complex systems, it is important to be able to describe them at different levels of abstraction. Abstraction is supported by disallowing or hiding an action from being used for communication with another process. Although the identity of a hidden action is removed from the observed behaviors of a process, their occurrences still affect the timed behavior of the process. For 2-way communication, $P \backslash a$ represents the process P with the actions a and $\bar{a}$ restricted; that is, the action a is not allowed to occur. The only way that a can happen is if it has already synchronized with $\bar{a}$ and thus converted into τ. For n-way communication, $P \backslash a$ represents the process P from which the occurrence of action a has been hidden. The effect is that the action a can no longer be used in communication with other processes. When considering time, the hidden action a is assumed to occur immediately since it can no longer be used for synchronization with another process.

Resource. It is a fact that the timed behavior of a process depends on the availability of resources needed to execute the process. For example, consider two processes P and Q that are ready to execute actions a and b, respectively. If they need the same resource for execution, which can be used by only one process at a time, the executions of a and b will have to be interleaved. On the other hand, if there are enough resources, they can be executed in parallel. Most existing real-time processes support the notion of resources implicitly in two extreme ways. One extreme is one-to-one assignment of processes to processors. Thus, if two processes are ready to communicate, the communication will not be delayed. This view of concurrency is known as *maximum parallelism.* The other extreme is to assume that there is only one processor and thus all executions of processes are interleaved. We take a more realistic view in ACSR, where we assume a limited number of active resources, each of which is capable of executing one action at a time.

Priority. Priorities are used in every practical real-time systems to provide timely responses through scheduling of limited resources. Some real-time process algebras implicitly support a limited notion of priority, as the execution of an action is favored over delay. However, many real-time process algebras lack the explicit notion of priority, making it impossible to model interrupts or real-time process schedulers.

Priority can be provided by associating a priority to an action or by supporting a prioritized choice operator. In the former, if a has higher priority than b and if both of them are ready as in $a.P + b.Q$, the action a is executed. In the latter, the urgency is represented by the positions of arguments to the choice operator. For example, $P +_{>} Q$ means that the alternative P has a higher priority than the alternative Q. Although the relative priorities of actions can be changed within a process with the prioritized choice operator, the "true dynamic priority" in which the priorities of actions can be reassigned according to the current state cannot be supported with the prioritized choice operator. The reason is that action priorities are determined syntactically through relative positions of the actions as arguments to the choice operator. ACSR supports priorities that are associated with actions, since this is the approach taken in the design of real-time operating systems. For example, in rate-monotonic scheduling, priorities are associated with periodic tasks, with priorities based on the tasks' relative frequencies [23]. Using the ACSR method, it is straightforward to model a task set which is scheduled under the rate-monotonic paradigm. On the other hand, this would be quite difficult using the indexed choice operator, since all of the action possibilities would have to be explicitly represented for each execution step.

Equational Laws. As pointed out in the introduction, one of the major reasons for using formal methods like process algebra is to facilitate verification of the correctness of a specification. The general strategy for showing that a specification S satisfies its requirement R is to represent both S and R as process terms P_S and P_R, respectively. Then, P_S and P_R are reduced to terms P'_S and P'_R, which are syntactically equivalent. The algebraic laws of a process algebra allow the transformation of one process term into another equivalent term. A sound and complete set of such laws forms the foundation for analysis technique in any process algebra.

These laws are based on a notion of equivalence between terms. There are many different equivalence relations such as bisimulation and strong bisimulation[17, 29], testing preorder[13], *etc.* The equivalence relation of ACSR is called ***prioritized strong bisimulation***, which is strong bisimulation on a prioritized transition system.

There are two features of process algebras that make them effective in analyzing large systems. The first is the presence of a hiding or restriction operator that allows one to abstract away unnecessary details. The second is that equality for the process algebra is also a congruence relation. This allows the substitution of one component with another equal component in large systems. Thus, a large system can be broken into simpler subsystems and then proved correct in a modular fashion.

3 The Computation Model

The executions of a process are defined by a labelled transition system. For example, a process P_1 may have the following behavior:

$$P_1 \xrightarrow{\alpha_1} P_2 \xrightarrow{\alpha_2} P_3 \xrightarrow{\alpha_3} \ldots$$

That is, P_1 first executes α_1 and evolves into P_2, which executes α_2, etc. P_i represents the process's state at the ith step of an execution, while α_i represents the ith step, or *action* taken in the execution. This is a common – almost generic – way of describing a process behavior. In a process algebra, however, the states P_i are typically described by a concrete syntax, i.e., a language. Further, there is a finite set of transition rules which infer the stepwise behavior of the process.[1]

In our algebra there are two types of actions: those which consume time, and those which are instantaneous. The time-consuming actions represent the progress of t time units of a global clock. These actions may also represent the consumption of resources, e.g., CPUs in the system configuration. On the other hand, the instantaneous actions (or *events*) provide a basic mechanism for synchronization and communication between concurrent processes.

This dual-approach is motivated by the behavior of concurrent processes written in Ada and related languages. For example, a point-to-point handshaking that takes place between two tasks (e.g., the instant when a server accepts a **select** guard) can be modeled using an instantaneous event. The resource requirements of a task can be modeled by a sequence of time-consuming actions.

As we show in this section, the two classes of actions have separate priority orderings. The reason for this follows from the two roles that priority can play in a real-time system. First, there is the type of priority that comes "from above," i.e., from a specification. This type is used to "break a tie" between two competing services, and is modeled in ACSR

[1] The technique of a structured transition system is not limited to process algebras; e.g. see [34].

by priority on instantaneous events. (In Ada this is called *preference control* between guards, whereas in Occam the **PRI ALT** statement is provided for a similar purpose.) The other type of priority is introduced "from below," by the system's real-time scheduler. Naturally, this is modeled by a priority relation on time-consuming actions.

Timed Actions. We consider a system to be composed of a finite set of serially reusable resources, denoted by $\mathcal{R}$. An action that consumes one "tick" of time is drawn from the domain $\mathbf{P}(\mathcal{R} \times \boldsymbol{N})$, with the restriction that each resource be represented at most once. As an example, the singleton action, $\{(r, p)\}$, denotes the use of some resource $r \in \mathcal{R}$ running at the priority level p. The action $\emptyset$ represents idling for one time unit, since all resources are inactive.

We use $\mathcal{D}_R$ to denote the domain of timed actions, and we let A, B, C range over $\mathcal{D}_R$. We define $\rho(A)$ to be the set of resources used by the action A; e.g., $\rho(\{(r_1, p_1), (r_2, p_2)\}) = \{r_1, r_2\}$. We also use $\pi_r(A)$ to denote the priority level of the use of the resource r in the action A; e.g., $\pi_{r1}(\{(r_1, p_1), (r_2, p_2)\}) = p_1$. By convention, if r is not in $\rho(A)$, then $\pi_r(A) = 0$.

Instantaneous Events. We call instantaneous actions *events*, which provide the basic synchronization in our process algebra. An event is denoted by a pair (a, p), where a is the *label* of the event, and p is its *priority*. Labels are drawn from the set $\mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau\}$, where if a is a given label, we say that $\bar{a}$ is its *inverse* label; i.e., $\bar{\bar{a}} = a$. As in CCS, the special identity label, τ, arises when two events with inverse labels are executed simultaneously.

We use $\mathcal{D}_E$ to denote the domain of events, and let e, f and g range over $\mathcal{D}_E$. We use $l(e)$ and $\pi(e)$ to represent the label and priority, respectively, of the event e.

Finally, the entire domain of actions is $\mathcal{D} = \mathcal{D}_R \cup \mathcal{D}_E$, and we let α and β range over $\mathcal{D}$.

4 The Syntax and Operational Semantics

The following grammar describes the syntax of processes:

$$\begin{array}{rcl} P & ::= & \mathrm{NIL} \mid A : P \mid (a,n).P \mid P + P \mid P \| P \mid \\ & & P \triangle^a_t (P,P,P) \mid [P]_I \mid P \backslash F \mid rec\, X.P \mid X \end{array}$$

NIL is a process that executes no action (i.e., it is initially deadlocked). There are two prefix operators, corresponding to the two types of actions. The first, $A : P$, executes a resource-consuming action A at the first time unit, and proceeds to the process P. On the other hand, $(a, n).P$, executes the instantaneous event (a, n), and proceeds to P. The difference here is that we consider no time to pass during the event occurrence. The

Choice operator $P+Q$ represents nondeterminism – either of the processes may be chosen to execute, subject to the constraints of the environment and preemption relation. The operator $P||Q$ is the parallel composition of P and Q.

The Scope construct $P \bigtriangleup^{a}_{t} (Q, R, S)$ binds the process P by a temporal scope [22], and incorporates both the features of timeouts and interrupts. We call t the *time bound*, where $t \in \mathbf{N} \cup \{\infty\}$ (i.e., t is either a non-negative integer or infinity). P executes for a maximum of t time units. The scope may be exited in a number of ways. First, if P successfully terminates within time t by executing an event labelled with $\bar{a}$, then control proceeds to the "success-handler" Q (here, a may be any label other than τ). On the other hand, R is a timeout exception-handler; that is, if P fails to terminate within time t, then control proceeds to R. Lastly, at any time while P is executing it may be interrupted by S, and the scope is then departed.

The Close operator, $[P]_I$, produces a process P that monopolizes the resources in $I \subseteq \mathcal{R}$. The Restriction operator, $P \backslash F$, limits the behavior of P. Here, no events with labels in F are permitted to execute. The process *rec* $X.P$ denotes standard recursion, allowing the specification of infinite behaviors.

The semantics is defined in two steps. First, we develop the *unconstrained* transition system, where a transition is denoted as $P \xrightarrow{A} P'$. Within "$\longrightarrow$" no priority arbitration is made between actions; rather, we subsequently refine "$\longrightarrow$" to define our prioritized transition system, "$\longrightarrow_\pi$."

4.1 The Structured Transition System

The two rules for the prefix operators are *axioms*; i.e., they have premises of *true*. There is one rule for a time-consuming action, and one for an instantaneous action.

$$\textbf{ActT}\ \frac{-}{A : P \xrightarrow{A} P} \qquad\qquad \textbf{ActI}\ \frac{-}{(a,n).P \xrightarrow{(a,n)} P}$$

For example, the process $\{(r_1,p_1),(r_2,p_2)\} : P$ simultaneously uses resources r_1 and r_2 for one time unit, and then executes P. Likewise, the process $(a,p).P$ executes the event "(a,p)," and proceeds to P.

The rules for Choice are identical for both timed actions and instantaneous events (and hence we use "α" as the label).

$$\textbf{ChoiceL}\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad\qquad \textbf{ChoiceR}\ \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

As an example, $(a,7).P\ +\ \{(r_1,3),(r_2,7)\} : Q$ may choose between executing the event $(a,7)$ or the time-consuming action $\{(r_1,3),(r_2,7)\}$. The former behavior is deduced from rule **ActI**, while the latter is deduced from **ActT**.

The Parallel operator provides the basic constructor for concurrency and communication. The first rule, **ParT**, is for two time-consuming transitions.

$$\textbf{ParT}\quad \frac{P \xrightarrow{A_1} P', Q \xrightarrow{A_2} Q'}{P\|Q \xrightarrow{A_1 \cup A_2} P'\|Q'} \quad (\rho(A_1) \cap \rho(A_2) = \emptyset)$$

Note that timed transitions are truly synchronous, in that the resulting process advances only if both of the constituents take a step. The condition $\rho(A_1) \cap \rho(A_2) = \emptyset$ mandates that each resource is truly sequential, and that only one process may use a given resource during any time step.

The next three laws are for event transitions. As opposed to timed actions, events may occur asynchronously (as in CCS and related interleaving models).

$$\textbf{ParIL}\quad \frac{P \xrightarrow{(a,n)} P'}{P\|Q \xrightarrow{(a,n)} P'\|Q} \qquad\qquad \textbf{ParIR}\quad \frac{Q \xrightarrow{(a,n)} Q'}{P\|Q \xrightarrow{(a,n)} P\|Q'}$$

$$\textbf{ParCom}\quad \frac{P \xrightarrow{(a,n)} P', Q \xrightarrow{(\bar{a},m)} Q'}{P\|Q \xrightarrow{(\tau,n+m)} P'\|Q'}$$

The first two rules show that events may be arbitrarily interleaved. The last rule is for two synchronizing processes; that is, P executes an event with the label a, while Q executes an event with the inverse label $\bar{a}$. Note that when the two events synchronize, their resulting priority is the sum of their constituent priorities.

Example 4.1 Consider the following two processes:

$$\begin{aligned} P &\stackrel{\text{def}}{=} ((a,3).P_1) \;+\; (\{(r_3,8)\} : P_2) \\ Q &\stackrel{\text{def}}{=} ((\bar{a},5).Q_1) \;+\; (\{(r_1,7)\} : P_2) \end{aligned}$$

The compound process $P\|Q$ admits the following four transitions:

$$\begin{array}{lll} P\|Q \xrightarrow{(a,3)} P_1\|Q & & [\text{by } \textbf{ParIL}] \\ P\|Q \xrightarrow{(\bar{a},5)} P\|Q_1 & & [\text{by } \textbf{ParIR}] \\ P\|Q \xrightarrow{(\tau,8)} P_1\|Q_1 & & [\text{by } \textbf{ParCom}] \\ P\|Q \xrightarrow{\{(r_1,7),(r_3,8)\}} P_2\|Q_2 & & [\text{by } \textbf{ParT}] \end{array}$$

Note than an event transition always executes before the next "tick" of the global clock. □

The construction of **ParCom** helps ensure that the *relative* priority ordering among events with the same labels remains consistent event after communication takes places. The following example shows how the ordering is preserved.

Example 4.2 Consider the following two processes.

$$\begin{array}{lcl} P & \overset{\text{def}}{=} & (a,2).P_1 \;+\; (a,3).P_2 \\ Q & \overset{\text{def}}{=} & (\bar{a},5).Q_1 \;+\; (\bar{a},3).Q_2 \end{array}$$

Thus, in P the second choice is preferred, while in Q the first choice is preferred. There are eight possible transitions for $P||Q$:

$$\begin{array}{ll} P||Q \xrightarrow{(a,2)} P_1||Q & \qquad P||Q \xrightarrow{(a,3)} P_2||Q \\ P||Q \xrightarrow{(\bar{a},5)} P||Q_1 & \qquad P||Q \xrightarrow{(\bar{a},3)} P||Q_2 \\ P||Q \xrightarrow{(\tau,7)} P_1||Q_1 & \qquad P||Q \xrightarrow{(\tau,5)} P_1||Q_2 \\ P||Q \xrightarrow{(\tau,8)} P_2||Q_1 & \qquad P||Q \xrightarrow{(\tau,6)} P_2||Q_2 \end{array}$$

While there are now six possible transitions labelled with τ, the addition of priorities in **ParCom** ensures that the original relative orderings are maintained. Note that the τ-transition with the highest priority is that associated with the derivative $P_2||Q_1$. These transitions had the highest priorities in their original constituent processes. □

The Scope operator possesses a total of five transition rules, which describe the various behaviors induced by a temporal scope. The first two rules show that as long as $t > 0$ and P fails to execute an event labelled with b, the executions of P continue.

$$\textbf{ScopeCT} \quad \frac{P \xrightarrow{A} P'}{P \,\Delta^b_t\, (Q,R,S) \xrightarrow{A} P' \,\Delta^b_{t-1}\, (Q,R,S)} \quad (t > 0)$$

$$\textbf{ScopeCI} \quad \frac{P \xrightarrow{(a,n)} P'}{P \,\Delta^b_t\, (Q,R,S) \xrightarrow{(a,n)} P' \,\Delta^b_t\, (Q,R,S)} \quad (\bar{a} \neq b, t > 0)$$

The **ScopeE** (for "end") shows how P can depart the temporal scope by executing an event labelled with $\bar{b}$. Upon exit, the label $\bar{b}$ is converted to the identity label τ (however, the same priority is retained).

$$\textbf{ScopeE} \quad \frac{P \xrightarrow{(\bar{b},n)} P'}{P \,\Delta^b_t\, (Q,R,S) \xrightarrow{(\tau,n)} Q} \quad (t > 0)$$

The next rule, **ScopeT** (for "timeout"), is applied whenever the scope times out; that is, when $t = 0$. At this point, control proceeds to the exception-handler R.

$$\textbf{ScopeT} \quad \frac{R \xrightarrow{\alpha} R'}{P \,\Delta^b_t\, (Q,R,S) \xrightarrow{\alpha} R'} \quad (t = 0)$$

Finally, ScopeI shows that the process S may interrupt (and kill) P while the scope is still active.

$$\textbf{ScopeI} \quad \frac{S \xrightarrow{\alpha} S'}{P \,\Delta^{b}_{t}\, (Q,R,S) \xrightarrow{\alpha} S'} \quad (t > 0)$$

Example 4.3 Consider the following specification: "Execute P for a maximum of 100 time units. If P executes an event labelled with $\bar{b}$ in that time, then stop the system. However, if P fails to finish within 100 time units, then start executing R. At any time during the execution of P, allow interruption by an event $(c,3)$, which will halt P, and initiate the interrupt-handler S." This system may be realized by the following process: $P \,\Delta^{b}_{100}\, (\mathrm{NIL}, R, (c,3).S)$. □

The Restriction operator defines a subset of instantaneous events that are excluded from the behavior of the system. This is done by establishing a set of labels, F ($\tau \notin F$), and deriving only those behaviors that do not involve events with those labels. Time-consuming actions, on the other hand, remain unaffected.

$$\textbf{ResT} \quad \frac{P \xrightarrow{A} P'}{P\backslash F \xrightarrow{A} P'\backslash F} \qquad\qquad \textbf{ResI} \quad \frac{P \xrightarrow{(a,n)} P'}{P\backslash F \xrightarrow{(a,n)} P'\backslash F} \quad (a, \bar{a} \notin F)$$

Example 4.4 Restriction is particularly useful in "forcing" the synchronization between concurrent processes. In Example 4.1, synchronization on a and $\bar{a}$ is not forced, since $P \| Q$ has transitions labelled with a and $\bar{a}$. On the other hand, $(P\|Q)\backslash\{a\}$ has only two transitions:

$$(P\|Q)\backslash\{a\} \xrightarrow{(\tau,8)} (P_1\|Q_1)\backslash\{a\} \quad \text{and} \quad (P\|Q)\backslash\{a\} \xrightarrow{\{(r_1,7),(r_3,8)\}} (P_2\|Q_2)\backslash\{a\}$$

In effect, the restriction declares that a and $\bar{a}$ define a "dedicated channel" between P and Q. □

While Restriction assigns dedicated channels to processes, the Close operator assigns dedicated resources. When a process P is embedded in a closed context such as $[P]_I$, we ensure that there is no further sharing of the resources in I. Assume that P executes a time-consuming action A. If A utilizes less than the full resource set I, the action is augmented with $(r,0)$ pairs for each unused resource $r \in I - \rho(A)$. The way to interpret Close is as follows. A process may idle in two ways – it may either release its resources during the idle time (represented by $\emptyset$), or it may hold them. Close ensures that the resources are held. (Instantaneous events are not affected.)

$$\textbf{CloseT} \quad \frac{P \xrightarrow{A_1} P'}{[P]_I \xrightarrow{A_1 \cup A_2} [P']_I} \quad (A_2 = \{(r,0) \mid r \in I - \rho(A_1)\})$$

$$\textbf{CloseI} \quad \frac{P \xrightarrow{(a,n)} P'}{[P]_I \xrightarrow{(a,n)} [P']_I}$$

The operator *rec* $X.P$ denotes recursion, allowing the specification of infinite behaviors.

$$\textbf{Rec} \quad \frac{P[rec\ X.P/X] \xrightarrow{\alpha} P'}{rec\ X.P \xrightarrow{\alpha} P'}$$

where $P[rec\ X.P/X]$ is the standard notation for substitution of $rec\ X.P$ for each free occurrence of X in P.

As an example, consider $rec\ X.(A : X)$, which indefinitely executes the resource-consuming action "A." By **ActT**, $A : (rec\ X.(A : X)) \xrightarrow{A} rec\ X.(A : X)$, so by **Rec**, $rec\ X.(A : X) \xrightarrow{A} rec\ X.(A : X)$.

4.2 Preemption and Prioritized Transitions

The prioritized transition system is based on *preemption*, which incorporates our treatment of synchronization, resource-sharing, and priority. The definition of preemption is straightforward. Let "$\prec$", called the *preemption relation*, be a transitive, irreflexive, binary relation on actions. Then for two actions α and β, if $\alpha \prec \beta$, we can say that "α is preempted by β." This means that in any real-time system, if there is a choice between executing either α or β, it will always execute β.

Definition 4.1 (Preemption Relation) *For two actions, α, β, we say that β preempts α ($\alpha \prec \beta$), if one of the following cases hold:*

(1) Both α and β are timed actions in $\mathcal{D}_R$, where

$$(\rho(\beta) \subseteq \rho(\alpha)) \wedge (\forall r \in \rho(\alpha).\pi_r(\alpha) \leq \pi_r(\beta)) \wedge (\exists r \in \rho(\beta).\pi_r(\alpha) < \pi_r(\beta))$$

(2) Both α and β are events in $\mathcal{D}_E$, where $\pi(\alpha) < \pi(\beta) \wedge l(\alpha) = l(\beta)$

(3) $\alpha \in \mathcal{D}_R$ and $\beta \in \mathcal{D}_E$, with $l(\beta) = \tau$ and $\pi(\beta) > 0$. □

Case (1) shows that the two timed actions, α and β, compete for common resources, and in fact, the preempted action α may use a superset of β's resources. However, β uses all the resources at least the same priority level as α (recall that $\pi_r(B)$ is, by convention, 0 when r is not in B). Also, β uses at least one resource at a higher level.

Case (2) shows that an event may be preempted by another event sharing the same label, but with a higher priority.

Finally, case (3) shows the single case in which an event and a timed action are comparable under "$\prec$." That is, if $n > 0$ in an event (τ, n), we let the event preempt any timed action.

Example 4.5 The following examples show some comparisons made by the preemption relation, "$\prec$."

a. $\{(r_1,2),(r_2,5)\} \prec \{(r_1,7),(r_2,5)\}$
b. $\{(r_1,2),(r_2,5)\} \not\prec \{(r_1,7),(r_2,3)\}$
c. $\{(r_1,2),(r_2,0)\} \prec \{(r_1,7)\}$
d. $\{(r_1,2),(r_2,1)\} \not\prec \{(r_1,7)\}$
e. $(\tau,1) \prec (\tau,2)$
f. $(a,1) \not\prec (b,2)$ if $a \neq b$
g. $(a,2) \prec (a,5)$
h. $\{(r_1,2),(r_2,5)\} \prec (\tau,2)$ □

We define the prioritized transition system "$\longrightarrow_\pi$," which simply refines "$\longrightarrow$" to account for preemption.

Definition 4.2 *The labelled transition system "$\longrightarrow_\pi$" is defined as follows: $P \xrightarrow{\alpha}_\pi P'$ if and only if*

a) $P \xrightarrow{\alpha} P'$ is an unprioritized transition, and

b) There is no unprioritized transition $P \xrightarrow{\beta} P''$ such that $\alpha \prec \beta$. □

Example 4.6 This example illustrates the use of synchronization and priorities to model a semaphore. The event label s_p represents the P operation of the semaphore and the event label s_v represents the V operation. The semaphore M is defined as follows:

$$M \stackrel{\text{def}}{=} \emptyset : M + (\bar{s_p},0).rec\ X.(\emptyset : X + (\bar{s_v},0).M)$$

To see how this works, let P_1 and P_2 be two processes that must execute a critical section using two robot arms, $CR = \{(left\_arm,1),(right\_arm,1)\}$ followed by non-critical section, NCR. We assume that the process P_1 has priority 1 and the process P_2 has priority 2.

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} \emptyset : P_1 + (s_p,1).CR : (s_v,1).NCR : P_1' \\ P_2 &\stackrel{\text{def}}{=} \emptyset : P_2 + (s_p,2).CR : (s_v,2).NCR : P_2' \\ S &\stackrel{\text{def}}{=} (P_1 \| P_2 \| M)\backslash\{s_p,s_v\} \end{aligned}$$

Before entering the critical section, each process must execute the event s_p. By applying the rules of the operational semantics, we see that there are only three unprioritized transitions that the system S can take:

$$
\begin{array}{ll}
(1) & S \xrightarrow{\emptyset} \; S \\
(2) & S \xrightarrow{(\tau,1)} \; (CR : (s_v, 1).NCR : P_1')\|P_2\|(rec\ X.(\emptyset : X + \bar{s_v}, 0).M))\backslash\{s_p, s_v\} \\
(3) & S \xrightarrow{(\tau,2)} \; (P_1\|CR : (s_v, 2).NCR : P_2')\|(rec\ X.(\emptyset : X + \bar{s_v}, 0).M))\backslash\{s_p, s_v\}
\end{array}
$$

Only transition (3) remains admitted by the prioritized transition system. This allows P_2 to proceed. From this point and until P_2 executes $(s_v, 2)$, both P_1 and M will have to idle, i.e., execute the $\emptyset$ transitions. The execution of $(s_v, 2)$ by P_2 releases the semaphore and subsequently allows P_1 to acquire it.

5 Bisimulation and Strong Equivalence

Our proof techniques are based on process equivalence, whereby we attempt to prove that one process P is equivalent to another process Q. Typically, P is a general operational specification of the problem, while Q is a more complicated implementation. While Q may have a more complicated syntax, the objective is to show that the two processes are operationally equivalent. In our paradigm, equivalence between processes is based on the concept of *strong bisimulation* [33], which compares the computation trees of the two processes.

Definition 5.1 *For a given transition system "$\rightsquigarrow$", any binary relation r is a* strong bisimulation *if, for $(P, Q) \in r$ and $\alpha \in \mathcal{D}$,*

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

In other words, if P (or Q) can take a step on α, then Q (or P) must also be able to take a step on α with both of the next states also bisimilar. There are some very obvious bisimulation relations; e.g. $\emptyset$ (which certainly adheres to the above rules) or syntactic identity. However, using the theory found in [27, 28, 29], it is straightforward to show that there exists a largest such bisimulation over "$\longrightarrow$," which we denote as "$\sim$." This relation is an equivalence relation, and is a congruence with respect to the operators [11]. Similarly, "$\sim_\pi$" is the largest strong bisimulation over "$\longrightarrow_\pi$," and we call it *prioritized strong equivalence.*

5.1 Laws

Definition 5.1 provides us with an immediate technique to show whether two processes are bisimulation-equivalent. Since "$\sim_\pi$" is the largest prioritized bisimulation, any other bisimulation is contained in it. One can show, for example, that $(A : \mathrm{NIL}) \ + \ \mathrm{NIL} \ \sim_\pi A : \mathrm{NIL}$ by finding a binary relation r such that:

1. $(\ (A : \mathrm{NIL}) \ + \ \mathrm{NIL}\ ,\ A : \mathrm{NIL}\ ) \in r$, and that

2. r is a bisimulation.

Since $r \subseteq\sim_\pi$, the two processes are then guaranteed to be equivalent. Obviously, in this case the following r will suffice:

$$r \ = \ \{(\ (A : \mathrm{NIL}) \ + \ \mathrm{NIL}\ ,\ A : \mathrm{NIL}\ )\} \ \cup \ \{(\mathrm{NIL}, \mathrm{NIL})\}$$

For more complicated processes, carrying out proofs in such a model-oriented fashion can be difficult indeed. Finding the right relation r is not a straightforward procedure. Fortunately, this can be automatically done for finite-state processes [8, 20].

Another alternative is to formulate a general set of equational laws which serve to characterize bisimulation equivalence. This approach is, after all, the way we manipulate arithmetic expressions; for example, without appealing to the theory of distributive rings, we frequently use the law, $3 * (x + y) = 3 * x + 3 * y$. The same approach is applied by process algebras in the domain of concurrent processes.

In fact, the main distinguishing feature of process algebras over other state-transition models is this: process algebras include a relatively small set of laws which can be used to prove equivalence in many different situations. As an example, our two processes above can be proved equivalent by a law which is common to almost every process algebra – $P + \mathrm{NIL} \ = \ P$. This law is true for *any* process P and thus, it can be used in many different instances (just as, in arithmetic, we use $x + 0 = x$).

Table 1 presents our equivalence-preserving laws for ACSR. In the sequel, wherever we use the equality symbol "=" in showing that two processes are equivalent, it means that we have used our laws to construct the proof. The bisimilarity of the processes follows from the soundness of the laws.

Note the use of the summation symbol $\sum$ in Par(3). The interpretation is as follows: Let I be an index set representing processes, such that for each $i \in I$, there is some corresponding process P_i. If $I = \{i_1, \ldots, i_n\}$, because of Choice(4) we are able to neglect parentheses and use the following notation:

$$\sum_{i \in I} P_i \overset{\mathrm{def}}{=} P_{i_1} + \ldots + P_{i_n}$$

and where $\sum_{i\in\emptyset} P_i \overset{\text{def}}{=} NIL$.

Par(3) is representative of many of the laws, in that its objective is to "undo" a constructor. That is, it reduces the "$\|$" operator to a simpler form – in this case, a process whose initial steps can be determined by the Prefix and Choice constructors.

Soundness. We claim that the ACSR proof system, $\mathcal{A}$, augmented with standard laws for substitution, is sound with respect to prioritized strong equivalence.

Theorem 5.1 *For any processes P and Q, if $\mathcal{A} \vdash P = Q$, then $P \sim_\pi Q$.*

For each axiom $P = Q$ in $\mathcal{A}$, one must construct a bisimulation to show that $P \sim_\pi Q$. Due to space limitation we leave out the details of the proof, which can be found in [5]. Here, however, we can present the simplest case to convey the proof's basic technique.

We claim that the following bisimulation r suffices to establish that $P + \text{NIL} \sim_\pi P$:

$$r \;=\; \{(P + \text{NIL}, P) \,|\, P \text{ is an ACSR process}\} \;\cup\; ID$$

where ID is simply the syntactic identity relation. Obviously, $P + \text{NIL} \xrightarrow{\alpha}_\pi Q$ if and only if $P \xrightarrow{\alpha}_\pi Q$, since NIL adds nothing to the behavior of a process. And since $(Q, Q) \in ID \subseteq r$, and ID is itself a bisimulation, Definition 5.1 is satisfied.

The soundness proofs for the other laws have a similar structure (though some are of significantly greater complexity). □

Completeness. We also claim that that the ACSR laws are complete for finite processes; i.e., processes without the "*rec*" operator. To carry out the proof sketch, we define a Prioritized Normal Form (PNF) as follows:

Definition 5.2 *An ACSR term is in* PNF *if it is of the form:*

$$\sum_{i\in I} A_i : P_i + \sum_{j\in J} (a_j, n_j).Q_j$$

where 1) for all indices l, m in $I \cup J$ we have that $\alpha_l \not\prec \beta_m$; and 2) each P_i and Q_j itself in PNF .

By induction on the length of a process term, it is routine to show that every finite process can be transformed to a prioritized normal form using $\mathcal{A}$ (see [5] for details). We note that Choice (5)-(7) are key to enforcing property 1); that is, whenever there are two subterms R and R' prefixed with α and α', respectively, such that $\alpha \prec \alpha'$, we use one of these laws to eliminate the subterm prefixed with α. □

Theorem 5.2 *For any finite processes P and Q, if $P \sim_\pi Q$, then $\mathcal{A} \vdash P = Q$.*

Choice(1)	$P + \mathrm{NIL} = P$
Choice(2)	$P + P = P$
Choice(3)	$P + Q = Q + P$
Choice(4)	$(P + Q) + R = P + (Q + R)$
Choice(5)	$A_1 : P_1 + A_2 : P_2 = A_2 : P_2$ if $A_1 \prec A_2$
Choice(6)	$(a_1, n_1).P_1 + (a_2, n_2).P_2 = (a_2, n_2).P_2$ if $(a_1, n_1) \prec (a_2, n_2)$
Choice(7)	$A : P + (\tau, n).Q = (\tau, n).Q$ if $n > 0$
Par(1)	$P \parallel Q = Q \parallel P$
Par(2)	$(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$
Par(3)	$(\sum_{i\in I} A_i : P_i \;+\; \sum_{i\in J}(a_j, n_j).Q_j) \parallel (\sum_{k\in K} B_k : R_k \;+\; \sum_{l\in L}(b_l, m_l).S_l)$ $= \left[\begin{array}{l} \sum_{i\in I, k\in K, \rho(A_i)\cap\rho(B_k)=\emptyset} (A_i \cup B_k) : (P_i \parallel R_k) \\ + \sum_{j\in J} (a_j, n_j).(Q_j \parallel (\sum_{k\in K} B_k : R_k \;+\; \sum_{l\in L}(b_l, m_l).S_l)) \\ + \sum_{l\in L} (b_l, m_l).((\sum_{i\in I} A_i : P_i \;+\; \sum_{j\in J}(a_j, n_j).Q_j)) \parallel S_l \\ + \sum_{j\in J, l\in L, a_j=\bar{b}_l} (\tau, n_j + m_l).(Q_j \parallel S_l) \end{array}\right.$
Scope(1)	$A : P \overset{b}{\Delta}_t (Q, R, S) = A : (P \overset{b}{\Delta}_{t-1} (Q, R, S)) + S$ if $t > 0$
Scope(2)	$(a, n).P \overset{b}{\Delta}_t (Q, R, S) = (a, n).(P \overset{b}{\Delta}_t (Q, R, S)) + S$ if $t > 0 \wedge \bar{a} \neq b$
Scope(3)	$(a, n).P \overset{b}{\Delta}_t (Q, R, S) = (\tau, n).Q + S$ if $t > 0 \wedge \bar{a} = b$
Scope(4)	$P \overset{b}{\Delta}_0 (Q, R, S) = R$
Scope(5)	$(P_1 + P_2) \overset{b}{\Delta}_t (Q, R, S) = P_1 \overset{b}{\Delta}_t (Q, R, S) \;+\; P_2 \overset{b}{\Delta}_t (Q, R, S)$
Scope(6)	$(\mathrm{NIL}) \overset{b}{\Delta}_t (Q, R, S) = S$ if $t > 0$
Res(1)	$\mathrm{NIL} \backslash F = \mathrm{NIL}$
Res(2)	$(P + Q) \backslash F = (P \backslash F) + (Q \backslash F)$
Res(3)	$(A : P) \backslash F = A : (P \backslash F)$
Res(4)	$((a, n).P) \backslash F = (a, n).(P \backslash F)$ if $a, \bar{a} \notin F$
Close(1)	$[\mathrm{NIL}]_I = \mathrm{NIL}$
Close(2)	$[P + Q]_I = [P]_I + [Q]_I$
Close(3)	$[A_1 : P]_I = (A_1 \cup A_2) : [P]_I$ where $A_2 = \{(r, 0) \mid r \in I - \rho(A_1)\}$
Close(4)	$[(a, n).P]_I = (a, n).[P]_I$
Rec(1)	$recX.P = P[recX.P/X]$

Table 1: The Set of ACSR Laws, $\mathcal{A}$

This theorem can be proved as follows: Using $\mathcal{A}$, P and Q can be transformed into $\hat{P}$ and $\hat{Q}$, respectively, where both $\hat{P}$ and $\hat{Q}$ are in PNF. So assume that

$$\hat{P} \equiv (\sum_{i \in I} A_i : P_i) \; + \; (\sum_{j \in J} (a_j, n_j).Q_j) \quad \text{and} \quad \hat{Q} \equiv (\sum_{k \in K} B_i : R_i) + (\sum_{l \in L} (b_l, m_l).S_l).$$

The remainder of the proof follows by induction on the maximum depth of $\hat{P}$ and $\hat{Q}$. If the maximum depth is 0 then $\hat{P} \equiv \hat{Q} \equiv NIL$, and we are done. Otherwise, assume $\hat{P} \xrightarrow{\alpha}_{\pi} P'$. (The proof for the other direction, e.g., starting with $\hat{Q}$, is identical). There are two possible cases:

case 1: α is a timed action A. Then $\hat{P} \xrightarrow{A}_{\pi} P'$, and for some $i \in I$, $A : P' \equiv A_i : P_i$. Since $\hat{P} \sim_{\pi} \hat{Q}$, $\hat{Q} \xrightarrow{A}_{\pi} Q'$. So for some $k \in K$, $A : Q' \equiv B_k : R_k$. Further, $P_i \sim_{\pi} R_k$, so by induction, $\mathcal{A} \vdash P_i = R_k$; thus, $\mathcal{A} \vdash A_i : P_i = B_k : R_k$. So for all $i \in I$, there is some $k \in K$ such that $\mathcal{A} \vdash A_i : P_i = B_k : R_k$, and by a similar argument the converse is true as well.

case 2: α is an event (a, n). The proof for this case is identical to case 1.

Finally, by using Choice to eliminate redundancies and to regroup terms, it follows that $\mathcal{A} \vdash \hat{P} = \hat{Q}$. The details of the full completeness proof can be found in [5]. □

5.2 Dining Philosophers Example

This example is derived from the well-known problem of the dining philosophers. To shorten the presentation, we assume only three philosophers and three forks. Each philosopher spends its time idling and eating. In order to eat, a philosopher needs to use two forks. The problem is to develop a system in which there is no deadlock and each philosopher always has a possibility to eat in the future. In addition, each philosopher must meet the following timing requirements. It takes one time unit to pick up the first fork, and an additional one time unit to pick up the second fork and to eat. After eating, each philosopher must idle for at least one time unit before attempting to eat again. Finally, the system will timeout and deadlock if a philosopher does not get his second fork within two time units after acquiring the first one.

In our specification we use the following derived operator:

$$P \rhd^{e}_{t} Q \overset{\text{def}}{=} P \, \Delta^{e}_{t} \, (\mathrm{NIL}, \mathrm{NIL}, Q)$$

That is, $P \rhd^{e}_{t} Q$ is the process that may execute P for t time units, and then deadlock. However, at any time during those t time units, Q may "jump in" and take control of execution. Note that we may use the transition rules for Scope to infer the behavior of this derived construct. Also, using the laws for Scope we can give their abbreviated versions for our new derived operator:

$$\begin{array}{ll}
A : P \rhd^b_t S = A : (P \rhd^b_{t-1} S) + S \text{ if } t > 0 & \text{by Scope(1)} \\
(a,n).P \rhd^b_t S = (a,n).(P \rhd^b_t S) + S \text{ if } t > 0 \wedge a \neq b & \text{by Scope(2)} \\
(a,n).P \rhd^b_t S = (\tau,n).\mathrm{NIL} + S \text{ if } t > 0 \wedge a = b & \text{by Scope(3)} \\
P \rhd^b_0 S = \mathrm{NIL} & \text{by Scope(4)} \\
(P_1 + P_2) \rhd^b_t S = P_1 \rhd^b_t S \; + \; P_2 \rhd^b_t S & \text{by Scope(5)}
\end{array}$$

We present two specifications: one that is incorrect due to deadlock and one that is correct. In both specifications, three philosophers are represented by processes D_0, D_1 and D_2, and three forks are represented by resources f_0, f_1 and f_2. We assume that for each $0 \leq i \leq 2$, the forks f_i and $f_{(i+1)mod3}$ are located to the left and right, respectively, of the philosopher D_i. The process D_i uses the event e_i to represent that the ith philosopher has finished eating.

First Specification. The philosopher process D_i, for each $0 \leq i \leq 2$, and the system S are specified as follows:

$$\begin{array}{rcl}
D_i & \stackrel{\text{def}}{=} & \emptyset : D_i + \{(f_i, 1)\} : (D'_i \rhd^{\mathrm{e}}_2 D''_i) \\
D'_i & \stackrel{\text{def}}{=} & \{(f_i, 1)\} : D'_i \\
D''_i & \stackrel{\text{def}}{=} & \{(f_i, 2), (f_{(i+1)mod3}, 2)\} : (\mathrm{e}_i, 0).\emptyset : D_i \\
S & \stackrel{\text{def}}{=} & [D_0 \parallel D_1 \parallel D_2]_{\{f_0, f_1, f_2\}}
\end{array}$$

The process D_i may idle for any length of time or pick up the left fork, f_i. While holding the left fork, it waits for the right fork, $f_{(i+1)mod3}$. If it gets the right fork within two time units, i.e., executes the action $\{(f_i, 2), (f_{(i+1)mod3}, 2)\}$ via interrupt, it signals the completion of eating through the event $(e_i, 0)$. Otherwise, it times out and behaves as NIL. The system S is specified as the three processes composed in parallel. In addition, S is closed over the resources f_0, f_1 and f_2 since they are used only by the three philosopher processes.

To simplify the subsequent discussion, we write F^J_I for the action using the fork(s) denoted by I at priority 1 and the fork(s) denoted by J at priority 2. For example,

$$F^{02}_1 = \{(f_0, 2), (f_1, 1), (f_2, 2)\}$$

In addition, we write F for $\{f_0, f_1, f_2\}$ and extend the closure notation to actions as follows:

$$[A]_I = A \cup \{(r, 0) | r \in I - \rho(A).\}$$

One of the possible transitions of the system, S, is

$$S \xrightarrow{F_{012}} [D'_0 \triangleright^{e}_{2} D''_0 \parallel D'_1 \triangleright^{e}_{2} D''_1 \parallel D'_2 \triangleright^{e}_{2} D''_2]_F$$

This transition corresponds to the case in which all of the philosophers pick up their left forks at the same time. At this point the system can only perform two additional F_{012} transitions and deadlock. Other transitions are not possible due to resource conflicts.

Second Specification. To prevent deadlock, we let each philosopher take the fork with the lower number first. The only difference between this solution and the previous one is the process D_2, which is defined as follows:

$$\begin{array}{lcl} D_2 & \overset{\text{def}}{=} & \emptyset : D_2 + F_0 : (D'_2 \triangleright^{e}_{2} D''_2) \\ D'_2 & \overset{\text{def}}{=} & F_0 : D'_2 \\ D''_2 & \overset{\text{def}}{=} & F^{02} : (e_2, 0).\emptyset : D_2 \end{array}$$

Note that the F_{012} transition that leads to deadlock in the previous system is no longer possible. We now prove the correctness of the new system S.

Expanding S with the definitions of D_i gives

$$\begin{array}{lcll} S & = & [& \emptyset : D_0 + F_0 : (D'_0 \triangleright^{e}_{2} D''_0) \\ & & \parallel & \emptyset : D_1 + F_1 : (D'_1 \triangleright^{e}_{2} D''_1) \\ & & \parallel & \emptyset : D_2 + F_0 : (D'_2 \triangleright^{e}_{2} D''_2)]_F \end{array}$$

It can then be rewritten by Par(3):

$$\begin{array}{lcll} S & = & [& (\emptyset : (D_0 \parallel D_1 \parallel D_2)) \\ & & + & (F_0 : (D'_0 \triangleright^{e}_{2} D''_0 \parallel D_1 \parallel D_2)) \\ & & + & (F_1 : (D_0 \parallel D'_1 \triangleright^{e}_{2} D''_1 \parallel D_2)) \\ & & + & (F_0 : (D_0 \parallel D_1 \parallel D'_2 \triangleright^{e}_{2} D''_2)) \\ & & + & (F_{01} : (D'_0 \triangleright^{e}_{2} D''_0 \parallel D'_1 \triangleright^{e}_{2} D''_1 \parallel D_2)) \\ & & + & (F_{01} : (D_0 \parallel D'_1 \triangleright^{e}_{2} D''_1 \parallel D'_2 \triangleright^{e}_{2} D''_2))]_F \end{array}$$

by Close(2):

$$\begin{array}{lcll} S & = & & [\emptyset : (D_0 \parallel D_1 \parallel D_2)]_F \\ & & + & [F_0 : (D'_0 \triangleright^{e}_{2} D''_0 \parallel D_1 \parallel D_2)]_F \\ & & + & [F_1 : (D_0 \parallel D'_1 \triangleright^{e}_{2} D''_1 \parallel D_2)]_F \\ & & + & [F_0 : (D_0 \parallel D_1 \parallel D'_2 \triangleright^{e}_{2} D''_2)]_F \\ & & + & [F_{01} : (D'_0 \triangleright^{e}_{2} D''_0 \parallel D'_1 \triangleright^{e}_{2} D''_1 \parallel D_2)]_F \\ & & + & [F_{01} : (D_0 \parallel D'_1 \triangleright^{e}_{2} D''_1 \parallel D'_2 \triangleright^{e}_{2} D''_2)]_F \end{array}$$

and by Close(3):

$$\begin{array}{rcl} S & = & \{(f_0,0),(f_1,0),(f_2,0)\} : [(D_0 \parallel D_1 \parallel D_2)]_F \\ & + & [F_0]_F : [(D'_0 \rhd^e_2 D''_0 \parallel D_1 \parallel D_2)]_F \\ & + & [F_1]_F : [(D_0 \parallel D'_1 \rhd^e_2 D''_1 \parallel D_2)]_F \\ & + & [F_0]_F : [(D_0 \parallel D_1 \parallel D'_2 \rhd^e_2 D''_2)]_F \\ & + & [F_{01}]_F : [(D'_0 \rhd^e_2 D''_0 \parallel D'_1 \rhd^e_2 D''_1 \parallel D_2)]_F \\ & + & [F_{01}]_F : [(D_0 \parallel D'_1 \rhd^e_2 D''_1 \parallel D'_2 \rhd^e_2 D''_2)]_F \end{array}$$

We apply preemption, i.e., Choice(5), to obtain:

$$\begin{array}{rcl} S & = & [F_{01}]_F : [(D'_0 \rhd^e_2 D''_0 \parallel D'_1 \rhd^e_2 D''_1 \parallel D_2)]_F \\ & + & [F_{01}]_F : [(D_0 \parallel D'_1 \rhd^e_2 D''_1 \parallel D'_2 \rhd^e_2 D''_2)]_F \end{array}$$

Let's write this as $S \overset{\text{def}}{=} [F_{01}]_F : B_1 + [F_{01}]_F : B_2$. The process B_1 can be written by the definitions of D'_0, D'_1, D_2:

$$B_1 = [(F_0 : D'_0) \rhd^e_2 D''_0 \parallel (F_1 : D'_1) \rhd^e_2 D''_1 \parallel \emptyset : D_2 + F_0 : (D'_2 \rhd^e_2 D''_2)]_F$$

by Scope(1):

$$B_1 = [F_0 : (D'_0 \rhd^e_1 D''_0) + D''_0 \parallel F_1 : (D'_1 \rhd^e_1 D''_1) + D''_1 \parallel \emptyset : D_2 + F_0 : (D'_2 \rhd^e_2 D''_2)]_F$$

by Par(3) and the definition of D''_1:

$$\begin{array}{rcl} B_1 & = & [\quad F_{01} : (D'_0 \rhd^e_1 D''_0 \parallel D'_1 \rhd^e_1 D''_1 \parallel D_2) \\ & & +\quad F^{12}_0 : (D'_0 \rhd^e_1 D''_0 \parallel (e_1,0).\emptyset : D_1 \parallel D_2)]_F \end{array}$$

by Close(2), Close(3) and Choice(5):

$$B_1 = F^{12}_0 : [(D'_0 \rhd^e_1 D''_0 \parallel (e_1,0).\emptyset : D_1 \parallel D_2)]_F$$

and by Par(3):

$$B_1 = F^{12}_0 : (e_1,0).[(D'_0 \rhd^e_1 D''_0 \parallel \emptyset : D_1 \parallel D_2)]_F$$

We let

$$B'_1 \overset{\text{def}}{=} [(D'_0 \rhd^e_1 D''_0 \parallel \emptyset : D_1 \parallel D_2)]_F$$

and obtain by Scope(1) and substituting with the definition of D''_0 and D_2:

$$\begin{array}{rll} B'_1 = & [& (F_0 : (D'_0 \rhd^e_0 D''_0) + F^{01} : (e_0,0).\emptyset : D_0) \\ & \parallel & \emptyset : D_1 \\ & \parallel & \emptyset : D_2 + F_0 : (D'_2 \rhd^e_2 D''_2)]_F \end{array}$$

This can then be rewritten by Par(3):

$$\begin{array}{rll} B'_1 = & [& F_0 : (D'_0 \rhd^e_0 D''_0 \parallel D_1 \parallel D_2) \\ & + & F^{01} : ((e_0,0).\emptyset : D_0 \parallel D_1 \parallel D_2)]_F \end{array}$$

and by Close(3), Choice(5), Par(3) and Close(4):

$$B_1' = [F^{01}]_F : (e_0, 0).[\emptyset : D_0 \parallel D_1 \parallel D_2]_F$$

Substituting the definitions of D_1 and D_2 gives

$$\begin{aligned} B_1' = [F^{01}]_F : (e_0, 0). \quad & [\emptyset : D_0 \\ & \parallel \emptyset : D_1 + F_1 : (D_1' \rhd_2^e D_1'') \\ & \parallel \emptyset : D_2 + F_0 : (D_2' \rhd_2^e D_2'')]_F \end{aligned}$$

Which can then be rewritten by Par(3):

$$\begin{aligned} B_1' = [F^{01}]_F : (e_0, 0). \quad & [(\emptyset : (D_0 \parallel D_1 \parallel D_2)) \\ & +(F_1 : (D_0 \parallel D_1' \rhd_2^e D_1'' \parallel D_2)) \\ & +(F_0 : (D_0 \parallel D_1 \parallel D_2' \rhd_2^e D_2'')) \\ & +(F_{01} : (D_0 \parallel D_1' \rhd_2^e D_1'' \parallel D_2' \rhd_2^e D_2''))]_F \end{aligned}$$

By Close(2), Close(3) and Choice(5) we obtain

$$B_1' = [F^{01}]_F : (e_0, 0).[F_{01}]_F : [(D_0 \parallel D_1' \rhd_2^e D_1'' \parallel D_2' \rhd_2^e D_2'')]_F$$

Or

$$B_1' = [F^{01}]_F : (e_0, 0).[F_{01}]_F : B_2$$

Therefore,

$$B_1 = F_0^{12} : (e_1, 0).[F^{01}]_F : (e_0, 0).[F_{01}]_F : B_2$$

A similar reasoning applied to B_2 gives

$$B_2 = F_0^{12} : (e_1, 0).[F^{02}]_F : (e_2, 0).[F_{01}]_F : B_1 \;+\; [F_1^{02}]_F : (e_2, 0).B_1$$

Since $S \overset{\text{def}}{=} [F_{01}]_F : B_1 + [F_{01}]_F : B_2$, it can easily seen that the second specification does not deadlock and that each process D_i will be able to execute $(e_i, 0)$ in the future as required by the problem statement. In addition, one can observe that the system is free of starvation although D_1 gets to eat more often than the other philosophers.

6 Related Work

In this section, we compare ACSR to other real-time process algebras and also review the notions of priority supported in other process algebras. Since there is a large amount of work in the area of real-time process algebra, we limit our comparison to work that is more closely related to ACSR. Unlike ACSR, none of these algebras support an explicit notion of resources or priorities. For the work on priority, we restrict our discussion to process algebras, although the notion of priority has been added to other formal models such as Petri nets. They are briefly reviewed in [12].

6.1 Real-Time Process Algebras

Hennessy and Regan [14] present a process algebra TPL (Temporal Process Language), which extends CCS by a timed action σ. The timed action σ denotes idling until the next clock cycle and thus is equivalent to the idle action $\emptyset$ of ACSR. The maximal progress is assumed in TPL, that is, if $P|Q$ can communicate, the communication is immediate; otherwise, it can be delayed and time progresses. For example, $a.P + b.Q$ can idle, but $(a.P + b.Q)|\bar{a}.Q$ cannot idle and τ (the result of communication of a and $\bar{a}$) must occur immediately.

Yi [38] develops Timed CCS, which is basically an extension of TPL to continuous time. The process $\epsilon(t).P$ behaves as P after idling t units of time, where t is a positive real number. The model includes one important property, called *persistency*, which means that if a process can perform an action a, it remains possible after any time delay. Because of this property, it is not possible to model deadline without employing a watchdog process.

Moller and Tofts [30] describes Temporal CCS. This algebra is definable over an arbitrary time domain but it is developed only for a discrete time domain in their paper. The process $(t).P$ behaves as P after exactly t units of time, and $\delta.P$ behaves as P, but is willing to wait any amount of time before actually proceeding. Unlike TPL or Timed CCS, the process $a.P$ does not let time pass without performing the action a. For timeout, Temporal CCS supports two types of choice operator: the strong choice $(+)$ and the weak choice $(\oplus)$. The process $P + Q$ behaves as either P or Q with the choice made at the time of the first action. Thus, any initial passage of time must be allowed by both P and Q. The process $P \oplus Q$ behaves like $P + Q$, except that if only one of the operands may allow delay greater than t units of time, then the other operand will be dropped from the computation at the occurrence of a delay of t time units. The weak choice can be used to specify timeout.

There exist a number of models that have incorporated the notion of time into untimed CSP. This has been accomplished by extending the language with timing constructs and providing denotational time-based semantics. Each of these formalisms, depending on its notion of equivalence, uses assorted combinations of timed traces, histories, failures, ready sets and/or divergences as part of behaviors. The meaning of a process term is the set of all possible behaviors. The best known real-time extension to CSP is the work by Reed and Roscoe (Timed CSP) [35]. They add time to CSP by introducing a delay operator *wait* t, where t is a positive real. To ensure bounded nondeterminism, the time between any two consecutive events of a sequential process is assumed to be greater than a predefined constant $\delta > 0$. Unlike other process algebras, the semantics of the language is given by a denotational semantics called the timed-failure model. Because of the dense time domain, the language is no longer axiomatizable. For verification, Davies and Schneider developed a proof system that can be used to show that $P sat f$, where the formula f is in timed-failure trace logic [9].

Nicollin and Sifakis [31] presents the Algebra of Timed Processes (ATP), which extends a combination of CCS and ACP with a unique feature, the unit-delay operator ($\lfloor - \rfloor$). The process $\lfloor P \rfloor (Q)$ behaves as P if P starts executing before the next time unit; otherwise, after the delay of one time unit, it behaves as Q. It is equivalent to the process

$$\emptyset : \mathrm{NIL}\ \triangle_1\ (\mathrm{NIL}, Q, P)$$

of ACSR. ATP supports both timed actions and instantaneous asynchronous events. It differs from ACSR in that there is no notion of resource and thus all timed actions are assumed to be resource compatible with one another. Similarly to ACSR, time progresses synchronously, whereas events have asynchronous semantics. In fact, the separation of timed actions and instantaneous events in ACSR is inspired by ATP.

6.2 The Semantics of Priority

In [2] Baeten, Bergstra and Klop add the notion of priority to a finite subset of ACP without the presence of τ-events. This is accomplished by the introduction of a partial order over actions, "$>$", as well as a priority operator, "θ." As an example, if $a > b$, then

$$\theta(ax + by + z) = \theta(ax + z).$$

Thus in the parlance of ACSR, we would say that a preempts b. In this light, a "θ-free" agent P would be interpreted under "$\longrightarrow$", whereas $\theta(P)$ would be interpreted using "$\longrightarrow_\pi$". One major difference between [2] and ACSR is that preemption is "greedy"; that is, in general $\theta(P) \mid \theta(Q)$ does not have the same meaning as $\theta(P \mid Q)$, where $\mid$ represents parallel composition. The reason for this fact is that the priority of the synchronous action, "$a|b$," does not depend on the priorities of its two constituent actions, "a" and "b."

An interesting result of [2] is that the axioms needed to characterize θ cannot be added to ACP's unprioritized axiom systems, and remain sound with respect to a ready or failure semantics. Instead, the finer-grained ready-trace semantics is introduced to give meaning to prioritized processes.

A bi-level priority semantics for CCS is treated in [7], in which events are divided into two subsets: those of low priority (e.g., τ, a, b), and those of high priority (e.g., $\underline{\tau}, \underline{a}, \underline{b}$). Events may synchronize only with inverses of the *same* priority, which limits the range of priorities to a two-element, total order. When synchronization occurs between two unprioritized events (e.g., a and $\overline{a}$), the result is the unprioritized τ. Similarly, when $\underline{a}$ and $\underline{\overline{a}}$ synchronize, the result is $\underline{\tau}$. This $\underline{\tau}$-event is the only preemptive action, which gives rise to the following law: for any unprioritized a,

$$\underline{\tau}.P\ +\ a.Q = \underline{\tau}.P$$

In [6], Camilleri and Winskel extend CCS with a prioritized choice operator. Akin to Occam's PRI ALT, this construct selects the input event of highest priority. Also concentrating on Occam, Barrett [3] provides prioritized semantics within the context of CSP. He proceeds to show that in certain contexts, the introduction of priority can preclude the necessity for fairness assumptions. Again, the emphasis in [3] is on guards at the receiving end of a channel.

7 Conclusions

We have described a timed process algebra called ACSR that supports the notions of resources and priorities. ACSR employs a synchronous semantics for resource-consuming actions that take time and an asynchronous semantics for events that are instantaneous. There is a single parallel operator that can be used to express both interleaving at the event level and lock-step parallelism at the action level.

ACSR's algebraic laws are derived from a term equivalence based on prioritized strong bisimulation, which incorporates a notion of preemption based on priority, synchronization and resource utilization. As illustrated with the dining philosophers example, these laws can be used to rewrite process terms in proving the correctness of a real-time system.

There are two areas of research that should be explored to extend the capability of ACSR. The first extension is to support dynamic priorities. ACSR supports only static priority; i.e., the priorities of actions and events cannot change during the execution of a process. Since modeling of many real-time scheduling algorithms, such as earliest deadline first, first-come-first-served, *etc.*, requires dynamic priorities, it would be useful to support dynamic priority in timed process algebras. This requires some method to capture the state information and then use that information in reassigning priorities. The second extension is to allow dense time so that a timed action can take an arbitrary non-zero amount of time.

We are currently implementing a toolkit based on ACSR, which includes a user interface, a rewrite system based on the proof system $\mathcal{A}$, a bisimulation checker and a model checker using RTL [19]. We have found that it is rather difficult to use ACSR for non-trivial examples without computer assistance for syntax-checking, and for carrying out analysis. This toolkit will allow us to evaluate the effectiveness of the algebraic approach such as ACSR for the specification and analysis of large, complex real-time systems.

Acknowledgement. The authors gratefully acknowledge the comments and suggestions made by the referees in improving the quality of this paper. The work described in this paper was significantly benefited from discussions with the members of Real-Time Group at the University of Pennsylvania.

References

[1] R. Alur and D. Dill. Automata for modeling real-time systems. In *Proc. of 17th ICALP, LNCS 443*, pages 322–335. Springer Verlag, 1990.

[2] J. Baeten, J. Bergstra, and J. Klop. Ready-Trace Semantics for Concrete Process Algebra with a Priority Operator. *Computer Journal*, 30(6):498–506, 1987.

[3] G. Barrett. The Semantics of Priority and Fairness in occam. In *Proc. of 5th Int. Conf. Math. Foundations of Programming Semantics, LNCS 442*. Springer-Verlag, 1990.

[4] J.A. Bergstra and J.W. Klop. Algebra of Communicating Processes with Abstraction. *Journal of Theoretical Computer Science*, 37:77–121, 1985.

[5] P. Brémond-Grégoire, J.Y. Choi, and I. Lee. The Soundness and Completeness of ACSR (Algebra of Communicating Shared Resources). Technical Report MS-CIS-93-59, Univ. of Pennsylvania, June 1993.

[6] J. Camilleri and G. Winskel. CCS with Priority Choice. In *Proc. of IEEE Symposium on Logic in Computer Science*, 1991.

[7] R. Cleaveland and M. Hennessy. Priorities in Process Algebras. *Information and Computation*, 87:58–77, 1990.

[8] R. Cleaveland, J. Parrow, and B. Steffen. A Semantics-Based Verification Tool for Finite-State Systems. In *Proc. of Protocol Specification, Testing, and Verification, IX*, pages 287–302. Elsevier Science Publishers B.V., 1990.

[9] J. Davies and S. Schneider. An Introduction to Timed CSP. Technical Report PRG-75, Oxford University Computing Laboratory, UK, August 1989.

[10] A. Gabrielian and M.K. Franklin. Multilevel Specification of Real-Time Systems. *Comm. of ACM*, 35(5):51–60, 1991.

[11] R. Gerber. *Communicating Shared Resources: A Model for Distributed Real-Time Systems.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1991.

[12] R. Gerber and I. Lee. A Resource-Based Prioritized Bisimulation for Real-Time Systems. Technical Report MS-CIS-90-69, University of Pennsylvania, Department of Computer and Information Science, September 1990. To appear in *Information and Computation.*

[13] M. Hennessy. *Algebraic Theory of Processes.* MIT Press Series in the Foundations of Computing. MIT Press, 1988.

[14] M. Hennessy and T. Regan. A Process Algebra for Timed Systems. Technical Report 5/91, Univ. of Sussex, UK, April 1991.

[15] T. Henzinger, Z. Manna, and A. Pnueli. Temporal Proof Methodologies for Real-Time Systems. In *Proc. of ACM Principles of Programming Languages*, 1991.

[16] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–676, August 1978.

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

[18] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* PhD thesis, Eindhoven University of Technology, 1991.

[19] F. Jahanian and A.K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[20] P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[21] R. Koymans. Specifying Message Passing and Time-Critical Systems with Temporal Logic. *Real-Time Systems*, 16(11), November 1990.

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

[23] C.L. Liu and J.W. Layland. Scheduling algorithms for multi-programming in a hard-real-time environment. *Journal of the ACM*, pages 46 – 61, January 1973.

[24] N. Lynch and H. Attiya. Using Mappings to Prove Timing Properties. Technical Report MIT/LCS/TM-412b, Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[25] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[26] M. Merritt, F. Modungo, and M. Tuttle. Time-Constrained Automata. In *CONCUR '91*, August 1991.

[27] R. Milner. *A Calculus for Communicating Systems.* LNCS 92, Springer-Verlag, 1980.

[28] R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

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

[30] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proc. of CONCUR '90*, pages 401–415. LNCS 458, Springer Verlag, August 1990.

[31] X. Nicollin and J. Sifakis. The Algebra of Timed Processes ATP: Theory and Application. Technical Report RT-C26, Institut National Polytechnique De Grenoble, November 1991.

[32] J.S. Ostroff and W.M. Wonham. Modelling, Specifying and Verifying Real-time Embedded Computer Systems. In *Proc. of IEEE Real-Time Systems Symposium*, pages 124–132, December 1987.

[33] D. Park. Concurrency and Automata on Infinite Sequences. In *Proc. of 5th GI Conference.* LNCS 104, Springer Verlag, 1981.

[34] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.

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

[36] F.B. Schneider, B. Bloom, and K. Marzullo. Putting Time into Proof Outlines. Technical Report TR-93-1333, Cornell University, March 1993.

[37] A.C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[38] Wang Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *Proc. of Int. Conf. on Automata, Languages and Programming*, July 1991.