Alcom-FT Technical Report Series
ALCOMFT-TR-02-139

A FAMILY OF RESOURCE-BOUND REAL-TIME PROCESS ALGEBRAS

Insup Lee
University of Pennsylvania, Philadelphia, PA (lee@cis.upenn.edu)

Jin-Young Choi
Korea University, Seoul, Korea (choi@formal.korea.ac.kr)

Hee Hwan Kwak
Synopsys Inc., Beaverton, OR (hkwak@synopsys.com)

Anna Philippou
University of Cyprus, Cyprus (annap@cs.ucy.ac.cy)

Oleg Sokolsky
University of Pennsylvania, Philadelphia, PA (sokolsky@saul.cis.upenn.edu)

Abstract *This paper describes three real-time process algebras, ACSR, PACSR and ACSR-VP. ACSR is a resource-bound real-time process that supports synchronous timed actions and asynchronous instantaneous events as well as the notions of resource, priority, exception, and interrupt. PACSR is a probabilistic extension of ACSR with resources that can fail and associated failure probabilities. ACSR-VP extends ACSR with value passing between processes and parameterized process definitions. This paper also provides three simple real-time system examples to illustrate the expressive power and analysis technique of each process algebra.*

Keywords: Real-time process algebra, probablistic process algebra, value-passing process algebra, schedulability analysis, real-time systems, resource-bound process algebra.

1. INTRODUCTION

Reliability in real-time systems can be improved through the use of formal methods for the specification and analysis of timed behaviors. Recently, there has been a spate of progress in the development of real-time formal methods. 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 family of resource-bound real-time process algebras that we have developed.

Process algebras, such as CCS [12], CSP [7], Acceptance Trees [5] and ACP [2], 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 [12]. 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 which enable the verification of a whole system by reasoning about its parts. Process algebras are being used widely in specifying and verifying concurrent systems.

Algebra of Communicating Shared Resource (ACSR) introduced by Lee *et. al.* [10], is a timed process algebra which can be regarded as an extension of CCS. 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. ACSR 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.

ACSR supports the notions of resources, priorities, interrupt, timeout, and process structure. The notion of real-time in ACSR is quantitative and discrete, and is accommodated using the concept of timed actions. Executing a timed action requires access to a set of resources and takes one unit of time. Resources are serially reusable, and access to them is governed by priorities. Similar to CCS, the execution of an event is instantaneous and never consumes any resource. The notion of communication is modeled using events through the execution of complementary events, which are then converted into an internal event. As with timed actions, priorities are also used to arbitrate the choice of several events that are possible at the same time. Although the concurrency model of CCS-like process algebras is based on interleaving semantics, ACSR includes interleaving semantics for events as well as lock-step parallelism for timed actions.

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 the same time unit. Idling of a process is treated as a special timed action that consumes no resources. The execution of a timed action is subject to availability of the resources used in the timed action. 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 direct the choice when several events are possible at the same time.

We have extended ACSR into a family of process algebras, GCSR [1], Dense-time ACSR [4], ACSR-VP [9], and PACSR [15]. GCSR is a graphical version of ACSR which allows the visual representation of ACSR processes. Dense-time ACSR is an extension of ACSR with dense time. ACSR-VP extends ACSR with value-passing capability so that arbitrary scheduling problems can be specified and analyzed. Probabilistic ACSR allows the modeling of resource failure with probabilities.

The rest of the paper is organized as follows. Section 2 describes the basic computation model of ACSR and explains its notions of events and timed actions. Section 3 overviews the syntax and semantics of ACSR and describes a simple scheduling example. Section 4 explains PACSR and extends the same scheduling example with probabilistic resource failure. Section 5 describes ACSR-VP and shows how parametric scheduling analysis can be done using basically the same schedule example.

2. THE COMPUTATION MODEL

In our algebra there are two types of actions: those which consume time, and those which are instantaneous. The time-consuming actions represent one "tick" of a global clock. These actions may also represent the consumption of resources, e.g., CPUs, devices, memory, batteries in the system configuration. In contrast, the instantaneous actions provide a synchronization mechanism between a set of concurrent processes.

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 $\mathbb{P}(\mathcal{R} \times \mathbb{N})$, 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 no resuable resource is consumed.

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 resource r in action A; e.g., $\pi_{r_1}(\{(r_1, p_1), (r_2, p_2)\}) = p_1$. 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. We assume a set of channels L. 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 $\mathcal{L} \cup \bar{\mathcal{L}} \cup \{\tau\}$, where for all $a \in L$ $a? \in \mathcal{L}$ and $a! \in \bar{\mathcal{L}}$. We say that $a?$ and $a!$ are *inverse* labels. As in CCS, the special identity label, τ, arises when two events with inverse labels are executed in parallel.

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

The executions of a process are defined by a timed labelled transition system (timed LTS). A timed LTS M is defined as $\langle \mathcal{P}, \mathcal{D}, \rightarrow \rangle$, where (1) $\mathcal{P}$ is a set of ACSR processes, ranged over by P, Q, (2) $\mathcal{D}$ is a set of actions, and (3) $\rightarrow$ is a labeled transition relation such that $P \xrightarrow{\alpha} Q$ if the process P may perform an instantaneous event or timed action α and then behave as Q.

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. It takes no time to execute an instantaneous event. A timed action however is executed for exactly one unit of time.

3. ACSR

The following grammar describes the syntax of ACSR processes.

$$
\begin{aligned}
P \quad ::= \quad & \mathrm{NIL} \mid (a, n).\, P \mid A{:}P \mid P + P \mid P \| P \mid \\
& P \bigtriangleup_t^a (P, P, P) \mid P \backslash F \mid [P]_I \mid P \backslash\backslash I \mid b \rightarrow P \mid C.
\end{aligned}
$$

The process NIL represents the inactive process. There are two prefix operators, corresponding to the two types of actions. The process $(a, n).\, P$ executes the instantaneous event (a, n) and proceeds to P. The process $A{:}P$ executes a resource-consuming action during the first time unit and proceeds to P. The process $P + Q$ represents a nondeterministic choice between the two summands. The process $P \| Q$ describes the concurrent composition of P and Q: the component processes may proceed independently or interact with one another while executing events, and they synchronize on timed actions.

The scope construct, $P \triangle_t^a (Q, R, S)$, binds the process P by a temporal scope and incorporates the notions of timeout and interrupts. We call t the *time bound*, where $t \in \mathbb{N} \cup \{\infty\}$ and require that P may execute for a maximum of t time units. The scope may be exited in one of three ways: First, if P terminates successfully within t time-units by executing an event labeled $a!$ where $a \in L$, then control is delegated to Q, the success-handler. Else, if P fails to terminate within time t then control proceeds to R. Finally, throughout execution of this process construct, P may be interrupted by process S.

In $P \backslash F$, where $F \subseteq L$, the scope of channels in F is restricted to process P, and thus, components of P may use these labels to interact with one another but not with P's environment. The construct $[P]_I$, $I \subseteq \mathcal{R}$, produces a process that reserves the use of resources in I for itself, extending every action A in P with resources in $I - \rho(A)$ at priority 0. $P \backslash\backslash I$ hides the identity of resources in I so that they are not visible on the interface with the environment. That is, the operator $P \backslash\backslash I$ binds all free occurrences of the resources of I in P. This binder gives rise to the sets of *free* and *bound resources* of a process P. Process $b \rightarrow P$ represents the conditional process: it performs as P if boolean expression b evaluates to *true* and as NIL otherwise. Process constant C with process definition $C \stackrel{\text{def}}{=} P$ allows standard recursion.

The Structured Transition System. The informal account of behavior just given is made precise via a family of rules that define the labeled transition relations on processes. The semantics is defined in two steps. First, we develop the *unconstrained* transition system, where a transition is denoted as $P \xrightarrow{\alpha} P'$. Within "$\rightarrow$" no priority arbitration is made between actions; rather, we subsequently refine "$\rightarrow$" to define our prioritized transition system, "$\rightarrow_\pi$." The precise semantics rules are omitted but can be found in [3].

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, β will always be executed.

There are three cases to consider [3]: The first case is for the two timed actions, α and β, that compete for common resources. Here, the preempted action α may use a superset of β's resources. However, β uses all the resources at least at the same priority level as α. Thus, for any resource r in $\rho(\alpha) - \rho(\beta)$, the priority of r in α must be zero in order that β may preempt α since $\pi_r(B)$ is, by convention, 0 when r is not in B. Also, β uses at least one resource at a higher level. For instance, $\{(r_1, 2), (r_2, 0)\} \prec \{(r_1, 7)\}$ but $\{(r_1, 2), (r_2, 1)\} \not\prec \{(r_1, 7)\}$.

The second case is for the two events with the same label. Here, an event may be preempted by another event sharing the same label, but with a higher priority. For example, $(\tau, 1) \prec (\tau, 2)$, $(a, 2) \prec (a, 5)$, and $(a, 1) \not\prec (b, 2)$ if $a \neq b$.

The third case is when an event and a timed action are comparable under "$\prec$." Here, if $n > 0$ in an event (τ, n), we let the event preempt any timed action. For instance, $\{(r_1, 2), (r_2, 5)\} \prec (\tau, 2)$, but $\{(r_1, 2), (r_2, 5)\} \not\prec (\tau, 0)$.
We define the prioritized transition system "$\rightarrow_\pi$," which simply refines "$\rightarrow$" to account for preemption.

Definition 1 *The labeled transition system "$\rightarrow_\pi$" is defined as follows: $P \xrightarrow{\alpha}_\pi P'$ if and only if (1) $P \xrightarrow{\alpha} P'$ is an unprioritized transition, and (2) There is no unprioritized transition $P \xrightarrow{\beta} P''$ such that $\alpha \prec \beta$.* □

Analysis of Real-Time Systems in ACSR. Within the ACSR formalism we can conduct two types of analysis for real-time scheduling: validation and schedulability analysis. Validation shows that a given specification correctly models the required real-time scheduling discipline, such as Rate Monotonic and Earliest-Deadline-First. Schedulability analysis determines whether or not a real-time system with a particular scheduling discipline misses any of its deadlines. The validation and schedulability analysis of a real-time system can be carried out using the *equivalence* of ACSR processes.

Equivalence between ACSR processes is based on the concept of *bisimulation* [14] which compares the computation trees of two processes. Using the theory found in [12], it is straightforward to show that there exists a largest such bisimulation over "$\rightarrow_\pi$," which we denote as "$\sim_\pi$." This relation is an equivalence relation, and is a congruence with respect to ACSR's operators [3].

When comparing processes, we often find that because different objectives were pursued in formulating the two process expressions (perhaps simplicity of expressions for one, and efficiency for the other), the internal synchronization actions of the two processes are not identical. Consequently, even though the two processes may display identical "external" behavior (i.e., non-τ event labels and timed action steps), there may be τ actions in one process that do not correspond directly with τ actions in the other. (Recall that synchronization replaces the complementary event labels with a single τ event.) For those situations where matching of external behaviors is sufficient a weaker form of equivalence, *weak bisimulation* [12], is used. It is straightforward to prove the existence of a largest weak bisimulation $\approx_\pi$ over "$\rightarrow_\pi$" in a manner analogous to the case for bisimulation. Weak bisimulation, $\approx_\pi$, is an equivalence relation (though not a congruence) for ACSR that compares observable behaviors of processes.

Example. Throughout the paper, we will use a simple example from the area of schedulability analysis. The example describes a set of periodic tasks scheduled according to the Rate Monotonic (RM) scheduling policy. This policy assigns static priorities to the tasks in the inverse proportion to their periods. As a syntactic convenience, we allow ACSR processes to be parameterized by a set of index variables. Each index variable is given a fixed range of values. This restricted notion of parameterization allows us to represent collections of similar processes concisely. For example, the parameterized process

$$P_t = t < 2 \rightarrow (a_t, t).P_{t+1}, t \in \{0..2\}$$

is equivalent to the following three processes:

$$P_0 = (a_0, 0).P_1, \quad P_1 = (a_1, 1).P_2, \quad P_2 = \mathrm{NIL}.$$

As shown in Section 5, the addition of parameterized process definition in ACSR-VP allows us to get rid of this kind of parameterization.

The example is constructed as follows. We have two tasks, $Task_1$ and $Task_2$. $Task_i$ has period p_i and execution time e_i. The deadline for each task is equal to its period. Both tasks share the same processor, modeled by the resource *cpu*. No other tasks use the processor.

Each $Task_i$ idles until it is awakened by the operating system by the $start_i$ event and starts competing for the processor. At each time unit, the task may either get access to the processor or, if it is preempted by a higher-priority task, it idles until the next time unit. Once the necessary amount (i.e., e_i) of execution time is accumulated, the task returns to the initial state and waits for the next period.

In order to detect missed deadlines, we also model the task dispatcher, which initiates the tasks according to their periods. For each $Task_i$, there is a process $Dispatch_i$, which sends the $start_i$ event to the respective task every p_i time units. If the task cannot accept the event - that is, if it has not completed its execution - the dispatcher deadlocks.

The complete specification is shown below. In the specification of a task, i is the task number and j is the accumulated execution time. We assume that the tasks have distinct periods and are ordered by decreasing periods, so we can use the task number as the priority for processor access.

$$
\begin{array}{rcll}
\mathit{System} & \stackrel{\text{def}}{=} & [(\mathit{Dispatch}_1 | \mathit{Dispatch}_2 | \mathit{Task}_1 & \\
 & & \ | \mathit{Task}_2) \backslash \{\mathit{start}_1, \mathit{start}_2\}]_{\{cpu\}} & \\
\mathit{Dispatch}_i & \stackrel{\text{def}}{=} & (\mathit{start}_i!, i).D_{i,0} & i = \{1,2\} \\
D_{i,k} & \stackrel{\text{def}}{=} & k < p_i \rightarrow \{\} : D_{i,k+1} & \\
 & + & k = p_i \rightarrow \mathit{Dispatch}_i & i = \{1,2\}, k = \{0, p_i\} \\
\mathit{Task}_i & \stackrel{\text{def}}{=} & (\mathit{start}_i?, 0).P_{i,0} + \{\} : \mathit{Task}_i & i = \{1,2\} \\
P_{i,j} & \stackrel{\text{def}}{=} & j < e_i \rightarrow (\{\} : P_{i,j} & \\
 & & \qquad\quad + \{(cpu, i)\} : P_{i,j+1}) & \\
 & + & j = e_i \rightarrow \mathit{Task}_i & i = \{1,2\}, j = \{0, e_i\}
\end{array}
$$

ACSR analysis techniques allow us to verify the schedulability of a system of tasks for fixed values of parameters e_i and p_i. The correctness criterion being that a resulting process does not deadlock can be checked either by deciding the behavioral equivalence of the process to the process that idles forever, or by performing reachability analysis on the state space of the process to search for deadlock states. For example, we considered two sets of tasks. The task set with parameters $e_1 = 2$, $p_1 = 5$, $e_2 = 1$, $p_2 = 2$ does not exhibit any deadlock, while the set $e_1 = 2$, $p_1 = 3$, $e_2 = 1$, $p_2 = 2$ has a deadlock and thus is not schedulable.

4. PROBABILISTIC ACSR

PACSR (Probabilistic ACSR) extends the process algebra ACSR by associating with each resource a probability. This probability captures the rate at which the resource may fail. Since instantaneous events in PACSR are identical to those of ACSR, we only discuss timed actions, which now can account for resource failure.

Timed Actions. As in ACSR, we assume that a system contains a finite set of serially reusable resources drawn from the set $\mathcal{R}$. We also consider set $\overline{\mathcal{R}}$ that contains, for each $r \in \mathcal{R}$, an element $\overline{r}$, representing the *failed* resource r. We write R for $\mathcal{R} \cup \overline{\mathcal{R}}$. Actions are constructed as in ACSR, but now can contain both normal and failed resources. So now the action $\{(r,p)\}$, $r \in \mathcal{R}$, cannot happen if r has failed. On the other hand, action $\{(\overline{r}, q)\}$ takes place with priority q given that resource r has failed. This construct is useful for specifying recovery from failures.

Resource Probabilities. In PACSR we associate each resource with a probability at which the resource may fail. In particular, for all $r \in \mathcal{R}$ we denote by $\mathsf{p}(r) \in [0,1]$ the probability of resource r being up, while $\mathsf{p}(\overline{r}) = 1 - \mathsf{p}(r)$ denotes the probability of r failing. Thus, the behavior of a resource-consuming process has certain probabilistic aspects to it which are reflected in the operational semantics of PACSR. For example, consider the process $\{(cpu, 1)\} : \mathrm{NIL}$, where resource cpu has probability of failure $1/3$, i.e., $\mathsf{p}(\overline{cpu}) = 1/3$. Then, with probability $2/3$, resource cpu is available and thus the process may consume it and become inactive, while with probability $1/3$ the resource fails and the process deadlocks.

Probabilistic Processes. The syntax of PACSR processes is the same as that of ACSR. The only extension concerns the appearance of failed resources in timed actions. Thus, it is possible on one hand to assign failure probabilities to resources of existing ACSR specifications and perform probabilistic analysis on them, and, on the other hand, to ignore failure probabilities and apply non-probabilistic analysis of PACSR specifications.

As with ACSR, the semantics of PACSR processes is given in two steps. At the first level, a transition system captures the nondeterministic and probabilistic behavior of processes, ignoring the presence of priorities. Subsequently, this is refined via a second transition system which takes action priorities into account.

The unprioritized semantics is based on the notion of a *world*, which keeps information about the state of the resources of a process. Given a set of resources $Z \subset \mathcal{R}$, the set of possible worlds involving Z is given by $\mathcal{W}(Z) = \{Z' \subseteq Z \cup \overline{Z} \mid x \in Z' \text{ iff } \overline{x} \notin Z'\}$, that is, it contains all possible combinations of the resources in Z being up or down. Given a world $W \in \mathcal{W}(Z)$, we can calculate the probability of W by multiplying the probabilities of every resource in W.

Behavior of a given process P can be given only with respect to the world P is in. A *configuration* is a pair of the form $(P, W) \in \mathsf{Proc} \times 2^{\mathsf{R}}$, representing a PACSR process P in world W. The semantics is given in terms of a labeled transition system whose states are configurations and whose transitions are either probabilistic or nondeterministic. We write S for the set of configurations.

The intuition for the semantics is as follows: for a PACSR process P, we begin with the configuration $(P, \emptyset)$. As computation proceeds, probabilistic transitions are performed to determine the status of resources which are immediately relevant for execution but for which there is no knowledge in the configuration's world. Once the status of a resource is determined by some probabilistic transition, it cannot change until the next timed action occurs. Once a timed action occurs, the state of resources has to be determined anew, since in each time unit resources can fail independently from any previous failures. Nondeterministic transitions (which can be events or actions) may be performed from configurations that contain all necessary knowledge regarding the state of resources.

We partition the set S into probabilistic configurations S_p and non-deterministic configurations S_n. A configuration (P, W) is included in S_n if every resource that can be used in a first step of P is included in W. Transitions for a configuration $(P, W) \in S_n$ are determined in the same way as in ACSR for P, except that a transition labeled by an action A can be taken if every resource $r \in \mathsf{R}$ that appears in A is also contained in W. The probabilistic transition relation takes probabilistic configurations into non-deterministic configurations.

We illustrate the rules of the semantics with the following example. Consider the process $P = \{(r_1, 1), (r_2, 1)\} : P_1 + (e?, 1).P_2$. The immediately relevant resources of P are $\{r_1, r_2\}$. From the probabilistic configuration $(P, \{r_1\})$, where we know that r_1 is up, but have no information about r_2, we have two probabilistic transitions that determine the state of r_2: $(P, \{r_1\}) \overset{\mathsf{p}(r_2)}{\longmapsto} (P, \{r_1, r_2\})$ and $(P, \{r_1\}) \overset{\mathsf{p}(\overline{r_2})}{\longmapsto} (P, \{r_1, \overline{r_2}\})$. Both of these configurations are nondeterministic since we have full information about the relevant resources. Further, $(P, \{r_1, r_2\})$ has two nondeterministic transitions: $(P, \{r_1, r_2\}) \xrightarrow{\{(r_1,1),(r_2,1)\}} (P_1, \emptyset)$ and $(P, \{r_1, r_2\}) \xrightarrow{(e?,1)} (P_2, \{r_1, r_2\})$. The other configuration allows only one transition: $(P, \{r_1, \overline{r_2}\}) \xrightarrow{(e?,1)} (P_2, \{r_1, \overline{r_2}\})$, since r_2 is failed. Note that a probabilistic transition always leads to a nondeterministic configuration. A nondeterministic transition may lead to either nondeterministic configuration or a probabilistic one.

Probabilistic Analysis Techniques. We have defined a probabilistic weak bisimulation [16], which allows us to compare observable behaviors of PACSR processes similar to the case of ACSR. In addition, probabilistic information embedded in the probabilistic transitions allows us to perform quantitative analysis of PACSR specifications. In particular, we can compute the probability of reaching a given state or a deadlocked state.

Example. We illustrate the utility of PACSR in the analysis of fault-tolerance properties by slightly extending the example of Section 3. We consider the same set of tasks running on a processor with an intermittent fault. At any time unit, the processor may be running, in which case the higher-priority task executes normally, or it may be down, in which case none of the tasks execute. We modify the specification of a task to add the alternative behavior where a task can perform an action that contains the failed *cpu* resource and does not increase its execution time.

$$
\begin{array}{rcll}
System & = & [(Dispatch_1 | Dispatch_2 | Task_1 & \\
 & & \;| Task_2) \backslash \{start_1, start_2\}]_{\{cpu\}} & \\
Dispatch_i & = & (start_i!, i).D_{i,0} & i = \{1,2\} \\
D_{i,k} & = & k < p_i \rightarrow \{\} : D_{i,k+1} & \\
 & + & k = p_i \rightarrow Dispatch_i & i = \{1,2\}, k = \{0, p_i\} \\
Task_i & = & (start_i?, 0).P_{i,0} + \{\} : Task_i & i = \{1,2\} \\
P_{i,j} & = & j < e_i \rightarrow (\{\} : P_{i,j} & \\
 & & \qquad\quad + \{(cpu, i)\} : P_{i,j+1} & \\
 & & \qquad\quad + \{(\overline{cpu}, i)\} : P_{i,j}) & \\
 & + & j = e_i \rightarrow Task_i & i = \{1,2\}, j = \{0, e_i\}
\end{array}
$$

We apply the probabilistic analysis to the task set we considered in Section 3: $e_1 = 2$, $p_1 = 5$, $e_2 = 1$, $p_2 = 2$. Even though the task set is schedulable under perfect conditions, in the presence of failures the tasks may still miss their deadlines. Given the probability of a processor failure, we can compute the probability that a deadline is missed. The following list of pairs show results of the experiments we ran. The first element of each pair is the cpu failure probability and the second is the probability of a missed deadline: {(0,0), (0.005, 0.025), (0.01, 0.050), (0.02, 0.100), (0.05, 0.250), (0.075, 0.367), (0.1, 0.473), (0.15, 0.650), (0.2, 0.780), (0.3, 0.926)}.

5. ACSR-VP

ACSR-VP (ACSR with Value Passing) extends the process algebra ACSR described in Section 3 by allowing values to be communicated along communication channels. In this section we present ACSR-VP concentrating on its value-passing capabilities.

We assume a set of variables X ranged over by x, y and a set of values V ranged over by v. Moreover, we assume a set $Expr$ of expressions (which includes arithmetic expressions) and we let $BExpr \subset Expr$ be the subset containing boolean expressions. We let e and b range over $Expr$ and $BExpr$, respectively, and we write $\vec{z}$ for a tuple $z_1, \ldots z_n$ of syntactic entities.

As in ACSR, ACSR-VP also has two types of actions: instantaneous events and timed actions. The notion of timed action is identical to that of ACSR. However, instantaneous events are extended to provide value passing in addition to synchronization. An event is denoted as a pair (i, e_p) representing execution of action i at priority e_p, where i ranges over the internal τ, the input event $c?x$, and the output event $c!e$.

The syntax of ACSR-VP processes is similar to that of ACSR except for $C(\vec{x})$.

$$\begin{array}{rcl} P & ::= & \mathrm{NIL} \mid (a, n).\, P \mid A : P \mid P + P \mid P \| P \mid \\ & & P \bigtriangleup_t^a (P, P, P) \mid P \backslash F \mid [P]_I \mid P \backslash\backslash I \mid b \rightarrow P \mid C(\vec{x}). \end{array}$$

In the input-prefixed process $(c?x, e).P$ the occurrences of variable x is bound. We write $\mathsf{fv}(P)$ for the set of free variables of P. Each process constant C has an associated definition $C(\vec{x}) \stackrel{\mathrm{def}}{=} P$ where $\mathsf{fv}(P) \subseteq \vec{x}$ and $\vec{x}$ are pairwise distinct. We note that in an input prefix $(c?x, e).P$, e should not contain the bound variable x, although x may occur in P.

An informal explanation of ACSR-VP constructs is similar to that ACSR. The semantics of ACSR-VP process is also defined as a labeled transition system, similarly to that of ACSR. It additionally makes use of the following ideas: Process $(c!e_1, e_2).P$ transmits the value obtained by evaluating expression e_1 along channel c, with priority the value of expression e_2, and then behaves like P. Process $(c?x, p).P$ receives a value v from communication channel c and then behaves like $P[v/x]$, that is, P with v substituted for variable x. In the concurrent composition $(c?x, p_1).P_1 \| (c!v, p_2).P_2$, the two components of the parallel composition may synchronize with each other on channel c resulting in the transmission of value v and producing an event $(\tau, p_1 + p_2)$.

Symbolic Transition System. Consider the simple ACSR-VP process $P \stackrel{\mathrm{def}}{=} (in?x, 1).(out!x, 1).\mathrm{NIL}$ that receives a value along channel in and then outputs it on channel out, and where x ranges over integers. According to traditional methods for providing semantic models for concurrent processes using transition graphs, process P is infinite branching, as it can engage in the transition $(in?n, 1)$ for every integer value n. Thus, standard techniques for analysis and verification of finite state systems cannot be applied to such processes. Several approaches, such as symbolic transition graphs and transition graphs with assignment, have been proposed to deal with this problem for various subclasses of value-passing processes [6, 11, 13, 8]. We now briefly explain how to represent symbolic graphs with assignment for ACSR-VP processes. We only give an overview of the model and we refer to [8] for a complete discussion.

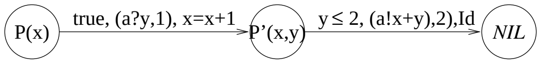
An SGA (Symbolic Graph with Assignment) is a rooted directed graph where each node n has an associated finite set of free variables $\mathsf{fv}(n)$ and each edge is labeled by a guarded action with assignment [11, 17]. Note that a node in SGA is an ACSR-VP term.

The notion of a *substitution*, which we also call *assignment*, is defined as follows. A *substitution* is any function $\theta: X \to Expr$, such that $\theta(x) \neq x$ for a finite number of $x \in X$. Given a substitution θ, the *support* (or *domain*) of θ is the set of variables $D(\theta) = \{x \mid \theta(x) \neq x\}$. A substitution whose support is empty is called the *identity substitution*, and is denoted by Id. When $|D(\theta)| = 1$, we use $[\theta(x)/x]$ for the substitution θ.

An SGA for ACSR-VP is a rooted directed graph where each node n has an associated ACSR-VP term and each edge is labeled by a boolean predicate, an action, and an assignment, (b, α, θ). Here we illustrate how to construct an SGA by an example. A set of rules for generating an SGA from an ACSR-VP term can be found in [8]. We use a transition $P \overset{b,\alpha,\theta}{\longmapsto} P'$ to denote that given the truth of boolean expression b, P can evolve to P' by performing actions α and putting into effect the assignment θ. Consider the following process.

$$\begin{array}{lcl} P(x) & \stackrel{\text{def}}{=} & (a?y, 1).P'(x+1, y) \\ P'(x, y) & \stackrel{\text{def}}{=} & (y \leq 2) \to (a!(x+y), 2).NIL \end{array}$$

The SGA for this process is shown below. Note how the value $x + 1$ is assigned along the edge from P to P'.



An informal interpretation of the above SGA is to view each process node as a procedure with its respective formal parameters. An edge coming into a node corresponds to a call to the procedure with the actual parameters supplied by the assignment labeling the edge and input events. If some of the variables are missing from the assignment, they are taken to be the same as the variable of the same name in the source node of the edge. After the process has been "called" in this way, it evaluates the guards on the outgoing transitions, applies the preemption relation to the enabled transitions and selects an one of the remaining transitions non-deterministically for the next step.

Symbolic Weak Bisimulation. The bisimulation relation for symbolic transition graphs is defined in terms of relations parametrized on boolean expressions, of the form $\simeq^b$, where $p \simeq^b q$ if and only if, for each interpretation satisfying boolean b, p and q are bisimilar in the traditional notion [8].

Let us compare the process P described above with the following process R:

$$
\begin{array}{lcl}
P(x) & \stackrel{\text{def}}{=} & (a?y, 1).P'(x+1, y) \\
P'(x, y) & \stackrel{\text{def}}{=} & (y \leq 2) \rightarrow (a!(x+y), 2).NIL \\
R(x') & \stackrel{\text{def}}{=} & (a?y', 1).R'(x, y') \\
R'(x', y') & \stackrel{\text{def}}{=} & (y' \leq 2) \rightarrow (a!(x'+y'+1), 2).NIL
\end{array}
$$

The prioritized SGA for R is a minor variation of the SGA for P. Applying the symbolic bisimulation algorithm for processes P and R, we obtain the following predicate equation system.

$$
\begin{array}{lcl}
X_{00}(x, x') & = & \forall z \forall z' X_{11}(z, z', x+1, x') \\
X_{11}(z, z', x, x') & = & z \leq 2 \rightarrow z' \leq 2 \wedge x + z = x' + z' + 1 \\
 & \wedge & z' \leq 2 \rightarrow z \leq 2 \wedge x' + z' + 1 = x + z
\end{array}
$$

This equation system can easily be reduced to the equation $X_{00}(x, x') \equiv x = x' + 1$, which allows us to conclude that $P(x)$ and $R(x')$ are bisimilar if and only if $x = x' + 1$ holds. In general, if we restrict to the domain of linear expressions, predicate equations obtained from the bisimulation algorithm can be solved using constraint logic programming and integer programming techniques [18].

Example. We revisit the example of Section 3 in order to conduct a more sophisticated, compared to ACSR, analysis allowed by ACSR-VP. With ACSR, the execution time and period of each task had to be fixed in order for the analysis to be performed. The symbolic semantics of ACSR-VP allows us to perform *parametric* analysis of processes. When we treat parameters of tasks as free variables of the specification, the scheduling problem can be restated in ACSR-VP as follows:

$$
\begin{array}{rcll}
System(e_1, e_2, p_1, p_2) & \stackrel{\text{def}}{=} & [(Dispatch_1(p_1) | Dispatch_2(p_2) | Task_1(e_1) & \\
 & & | Task_2(e_2)) \backslash \{start_1, start_2\}]_{\{cpu\}} & \\
Dispatch_i(p) & \stackrel{\text{def}}{=} & (start_i!, i).D_i(0, p) & i = \{1, 2\} \\
D_i(k, p) & \stackrel{\text{def}}{=} & k < p \rightarrow \{\} : D_i(k+1, p) & \\
 & + & k = p \rightarrow Dispatch_i(p) & i = \{1, 2\} \\
Task_i(e) & \stackrel{\text{def}}{=} & (start_i?, 0).P_i(0, e) + \{\} : Task_i(e) & i = \{1, 2\} \\
P_i(j, e) & \stackrel{\text{def}}{=} & j < e \rightarrow (\{\} : P_i(j, e) & \\
 & & \quad + \{(cpu, i)\} : P_i(j+1, e)) & \\
 & + & j = e \rightarrow Task_i(e) & i = \{1, 2\}
\end{array}
$$

We can now perform parametric analysis of *System* for some or all of its parameters. Here we consider the process $System(2, 1, p_1, p_2)$ Omitting the SGA for the example, we show the predicate equation system:

$$
\begin{array}{ll}
X_1(p_1,p_2) & = X_2(0,0,p_1,p_2) \\
X_2(j_{21},j_{22},p_1,p_2) & = X_3(0,0,j_{21},j_{22},p_1,p_2) \\
X_3(j_{11},j_{12},j_{21},j_{22},p_1,p_2) = & \\
\quad (j_{12} < p_1) \wedge (j_{21} < 1) \wedge (j_{22} < p_2) \wedge X_3(j_{11}, j_{12}+1, j_{21}+1, j_{22}+1, p_1, p_2) & \\
\quad +(j_{12} < p_1) \wedge (j_{11} < 2) \wedge (j_{22} < p_2) \wedge X_3(j_{11}+1, j_{12}+1, j_{21}, j_{22}+1, p_1, p_2) & \\
\quad +(j_{12} < p_1) \wedge (j_{11} = 2) \wedge (j_{22} < p_2) \wedge (j_{21} = 1) \wedge X_3(j_{11}, j_{12}+1, j_{21}, j_{22}+1, p_1, p_2) & \\
\quad +(j_{12} = p_1) \wedge (j_{11} = 2) \wedge X_3(0,0,j_{21},j_{22},p_1,p_2) & \\
\quad +(j_{22} = p_2) \wedge (j_{21} = 1) \wedge X_3(j_{11},j_{12},0,0,p_1,p_2) &
\end{array}
$$

We have the additional condition $p_2 < p_1$, which comes from the assumption made in Section 3 that processes are sorted by decreasing execution time. When we solve the system of equations under this condition, the set of values for the parameters is given by the pairs of integers (p_1, p_2) satisfying $p_1 > 3, p_2 > 1, p_1 > p_2$.

6. SUMMARY AND CURRENT WORK

We have presented three resource-bound real-time process algebras: ACSR, PACSR and ACSR-VP. ACSR employees a synchronous semantics for resource-consuming actions that take time and an asynchronous semantics for events that are instantaneous. ACSR was developed to handle schedulability analysis in a process-algebraic setting. PACSR supports the notion of probabilistic resource failures, whereas ACSR-VP extends ACSR with value-passing capability during communication and parameterized process definition. To illustrate their features, we have described and analyzed simple real-time systems using ACSR, PACSR, and ACSR-VP.

As mentioned in the introduction section, there are two more formalisms in the family of resource-bound real-time process algebras. One formalism is for visual specification of ACSR and the other is ACSR with dense time. We are currently developing a resource-aware process algebra to capture the notion of power consumption and resource constraints of embedded systems.

Acknowledgments. This research was supported in part by ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, DARPA ITO MOBIES F33615-00-C-1707, NSF CCR-9988409, NSF CCR-0086147, NSF CISE-9703220, and ONR N00014-97-1-0505 and by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

REFERENCES

[1] H. Ben-Abdallah. *GCSR: A Graphical Language for the Specification, Refinement and Analysis of Real-Time Systems.* PhD thesis, Department of Computer and

Information Science, University of Pennsylvania, 1996. Tech Rep IRCS-96-18.

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

[3] P. Brémond-Grégoire, J.Y. Choi, and I. Lee. A complete axiomatization of finite-state acsr processes. *Information and Computation*, 138(2):124–159, Nov 1997.

[4] P. Brémond-Grégoire and I. Lee. Process Algebra of Communicating Shared Resources with Dense Time and Priorities. *Theoretical Computer Science*, 189:179–219, 1997.

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

[6] M. Hennessy and H. Lin. Symbolic bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

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

[8] Hee-Hwan Kwak, Jin-Young Choi, Insup Lee, and Anna Philippou. Symbolic weak bisimulation for value-passing calculi. Technical Report MS-CIS-98-22, University of Pennsylvania, Department of Computer and Information Science, 1998.

[9] H.H. Kwak, I. Lee, A. Philippou, J.Y. Choi, and O. Sokolsky. Symbolic schedulability analysis of real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium.* IEEE Computer Society Press, 1998.

[10] I. Lee, P. Brémond-Grégoire, and R. Gerber. A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994.

[11] H. Lin. Symbolic graphs with assignment. In V.Sassone U.Montanari, editor, *Proceedings CONCUR 96*, volume 1119 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 1996.

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

[13] P. Paczkowski. Characterizing bisimilarity of value-passing parameterised processes. In *Proceedings of the Infinity Workshop on Verification of Infinite State Systems*, pages 47–55, 1996.

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

[15] A. Philippou, R. Cleaveland, I. Lee, S. Smolka, and O. Sokolsky. Probabilistic resource failure in real-time process algebra. In *Proc. of CONCUR'98*, pages 389–404. LNCS 1466, Springer Verlag, September 1998.

[16] A. Philippou, O. Sokolsky, and I. Lee. Weak bisimulation for probabilistic systems. In *Proc. of CONCUR'00*, pages 334–349. LNCS 1877, Springer Verlag, August 2000.

[17] J. Rathke. *Symbolic Techniques for Value-passing Calculi.* PhD thesis, University of Sussex, 1997.

[18] R. Saigal. *Linear Programming : A Modern Integrated Analysis.* Kluwer Academic Publishers, 1995.