Process Algebraic Modeling and Analysis of Power-Aware Real-Time Systems

Insup Lee, Anna Philippou, and Oleg Sokolsky

Abstract. The paper describes a unified formal framework for designing and reasoning about power-constrained, real-time systems. The framework is based on *process algebra*, a formalism which has been developed to describe and analyze communicating, concurrent systems. The proposed extension allows the modeling of probabilistic resource failures, priorities of resource usages, and power consumption by resources within the same formalism. Thus, it is possible to evaluate alternative power-consumption behaviors and tradeoffs under different real-time schedulers, resource limitations, resource failure probabilities, etc. This paper describes the modeling and analysis techniques, and illustrates them with examples, including a dynamic voltage-scaling algorithm.

1. Introduction

In recent years, there have been great technological advances in wireless communication and mobile computing. These advances have given rise to sophisticated embedded devices (e.g., PDA, cell phones, and smart sensors) and wireless network infrastructures that are becoming widely available. In addition, new applications with powerful functionalities are being developed to meet the ever-increasing demand by the users. A serious limitation of the mobile embedded devices is the battery life available to them. Although a great deal of power-intensive computation has to be performed to carry out application-specific functionalities, such as video streaming, this has to be done on a limited amount of power. To cope with this fact, a number of power-aware algorithms and protocols have been proposed aiming to make energy savings by dynamically altering the power consumed by a processor while still achieving the required behavior. However, in time-constrained applications often found in embedded systems, applying power-saving techniques can lead to serious problems. This is because changing the power available to tasks can affect their execution time which may lead to violation of their timing constraints and other desirable properties. A challenge presented by such systems is the development of robust algorithms that incorporate power-saving techniques and task management without sacrificing timing and performance guarantees. An example of such a proposal can be found in [1].

This paper describes a unified formal framework for designing and reasoning about power-constrained, real-time systems. The framework we propose is based on *process algebra*, a formalism which has been developed to describe and analyze communicating concurrently-executing systems, and is called P^2ACSR (Power-aware, Probabilistic ACSR). Process algebras are based on the premises that the two most essential notions in understanding complex dynamic systems are *concurrency* and *communication*. The most salient aspect of process algebras is that they support the *modular* specification and verification of systems, and thus, it is possible to analyze a whole system by reasoning about its parts. Process algebras are being used widely in specifying and verifying concurrent systems and they also have been extended to include the notions such as time and probability to facilitate the modeling of real-world systems.

The process algebra P^2ACSR extends our previous work on formal methods for real-time [2] and probabilistic systems [3] by incorporating the ability to reason about power consumption. The Algebra of Communicating Shared Resources (ACSR) [2] is a timed process algebra which represents a real-time system as a collection of concurrent processes. 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. That is, each process can engage in two kinds of activities: communication with other processes by instantaneous *events* and computation by timed *actions*. Executing a timed action requires access to a set of resources and takes a non-zero amount of time measured by an implicit global clock. Resources are serially reusable. The notion of a resource, which is essential in modeling of real-time

systems, additionally provides an important abstraction mechanism for capturing various aspects of systems behavior. One such aspect is the failure of physical devices: in PACSR [3], the probabilistic extension of ACSR, resources are extended with failure probabilities. In P^2ACSR, the resource model of PACSR is further extended to capture power consumption. Resources in P^2ACSR specifications are accompanied with information about the amount of power consumed by each resource use. Thus, for each execution step requiring access to a set of power-consuming resources, it is possible to compute the power consumed by the step and similarly for a sequence of such steps.

To be able to understand and analyze the timed behaviors and the power-consumption characteristics of P^2ACSR specifications, it is important that P^2ACSR has precisely defined semantics. The semantics needs to capture probabilistic behavior of the model, which is present due to resource failure, as well as nondeterministic behavior that reflects the possibility for a set of events to be enabled simultaneously from one or more processes. We have defined the operational semantics of P^2ACSR by a set of rules that can be used to derive all possible behaviors of a P^2ACSR process. Behaviors, in turn, are captured as *labeled concurrent Markov chains* [4], which are transition systems containing both probabilistic and nondeterministic execution steps.

Once a specification is written and its behaviors understood, the next step is to analyze whether or not the specified model satisfies required properties. We describe analysis methods that can be carried out on this model by extending model-checking techniques to allow reasoning about power consumption properties. Temporal logics are commonly used to express high-level requirements of systems including probabilistic criteria that apply to fragments of systems executions. We associate power-consumption constraints with temporal operators. For example, given a communication protocol in which a sender inquires about the readiness of a receiver and sends data after obtaining an acknowledgement, a useful property to check would be that this exchange happens correctly without consuming more than an acceptable level of power of the sender's processing unit. To further enable power-consumption calculations, we employ a technique that computes bounds on a system's power consumption. We illustrate the formalism and analysis techniques using some simple examples, including a dynamic voltage-scaling algorithm for real-time, power-aware systems [1]. In this example, we use resources to model the power-consuming processing unit which can be used at different power levels on different occasions. Furthermore, we model uncertainties in task execution times using the notion of probabilistic behaviors supported by P^2ACSR.

The rest of the paper is organized as follows: the next section explains the P^2ACSR framework, by introducing the syntax, providing informal semantics and presenting some examples. Section 3 describes analysis techniques for P^2ACSR processes, and Section 4 presents the case study in which a power-aware real-time scheduling algorithm is modeled and analyzed. We conclude with some final remarks and discussion of further work.

2. The Framework

Process algebra theory is now well established as a powerful mathematical model of concurrency, allowing the representation and analysis of communicating, concurrent systems and algorithms. Its basic entities are *processes* and *channels*, the former being the means for describing parts of a system acting concurrently and independently of other parts, and the latter being the points of interaction and synchronization amongst processes. The operators of process algebra, including operators such as sequential and parallel composition, allow for hierarchical description of systems, an invaluable feature for specification and compositional reasoning and analysis.

In our framework we extend this view of concurrent systems and include as a basic entity that of a *resource*. In particular, we consider a system to contain a finite set of reusable resources. Resources can be used to model physical entities, such as processor units and communication channels, or to abstract notions such as semaphores and message arrival. We allow resources to be associated with various attributes to capture aspects of resource-constrained computation. Such attributes can be associated to a resource itself or separately to each resource use. Specifically, we employ the following attributes:

- *Probability of failure*: We assume that resources of a system may fail during computation and associate to a resource a probability capturing the rate at which the resource may fail. A failure may correspond to a physical failure of a resource, such as a failure of a communication link, or the failure of some abstract condition, for example no message arrival when one was expected. Thus given a resource r we write $\pi(r)$ for the probability of r being up, consequently, r fails with probability $1 - \pi(r)$. To enable the specification of recovery from failures, we introduce the notion of a *failed* resource: for any resource *r* we write $\bar{r}$ for the failed resource *r* that can be used whenever resource *r* fails. This attribute remains constant throughout a system specification.
- *Priority*: Our formalism views a real-time system as a set of communicating processes sharing a set of serially-reusable resources. Since no more than a single system component can use a resource simultaneously, we choose to arbitrate the contention for the use of resources according to priorities. Thus, we associate to each resource use the priority level of the resource request.
- *Power consumption*: In order to reason about power consumption in distributed settings, we assume that the set of resources is partitioned into a finite set of disjoint classes, each class corresponding to a distinct power source which can provide a limited amount of power at any given time and for any period of time. Thus for each resource use we can specify the rate of power consumption required for that use and reason about the power consumption of a system.

Like its predecessors ACSR and PACSR, processes in P^2ACSR can engage in two types of activities, *instantaneous events* and *timed actions*. These may arise as follows:

Instantaneous events. Instantaneous events are the basic communication and synchronization primitives of the process algebra. They involve the use of a channel at some priority and they are denoted as a pair of the form *(a,p)* where *a* is the label of the event, the channel, and *p*, a natural number, the priority at which the channel is used. Labels are drawn from some set *L* where we assume that for each label *a* in *L* its *inverse* label $\bar{a}$ is also in *L*. The special label τ also assumed to be in *L* is called the *silent* event arises when two events with inverse label are executed concurrently. Events are assumed to take no time. We let *a, b* to represent labels.

Timed Actions. The second activity a process can engage in is that of a timed action. This involves the usage of a set of resources each at a certain priority and power-consumption level. Timed actions are assumed to consume one unit of time and are denoted by a finite set of triples of the form $\{(r_1, p_1, c_1),\ldots, (r_n, p_n, c_n)\}$ representing the intention of using each resource r_i at priority level p_i and power consumption level c_i. Action $\varnothing$ represents idling for one unit of time since no resource is consumed. We let *A, B,…* to represent timed actions.

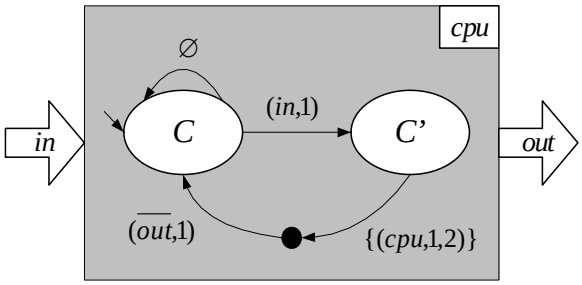
An example of a timed action is given by {(*cpu*,3,1),(*msg*,1,0)}. This action takes one unit of time, and uses resource *cpu* representing a processor unit, at priority level 3 and power consumption level 1. The processor can fail with probability $\pi(cpu)$. This action also assumes that the processor receives a message to continue its processing, represented by resource *msg*. The fact that this message may or may not arrive is modeled as the failure of resource *msg*. This is not a physical failure, but rather a failed assumption. The action {(*cpu*,3,1),(*msg*,1,0)} takes place assuming that none of the resources *cpu* and *msg* fail. On the other hand, action {(*cpu*,3,1),($\overline{msg}$,1,0)} takes place given that resource *msg* fails and resource *cpu* does not. So, for example, assuming that resources *cpu* and *msg* have probabilities of failure 0 and 1/3, respectively, that is, $\pi(cpu)=1$ and $\pi(msg) = 2/3$, then action {(*cpu*,3,1),($\overline{msg}$,1,0)} takes place with probability $\pi(cpu)\cdot\pi(\overline{msg})=1/3$ and fails with probability 2/3.

Processes. We let *P*, *Q* range over processes and we assume a set of process constants each with an associated definition of the kind $X \stackrel{def}{=} P$. The following grammar describes the syntax of P^{2}ACSR processes:

$$P \quad ::= \quad NIL \mid (a,p).\,P \mid A{:}P \mid b{\rightarrow}P \mid P + Q \mid P \,||\, Q \mid P\backslash F \mid [P]_I \mid rec\ X.\,P \mid X$$

Process *NIL* represents the inactive process. There are two prefix operators, corresponding to the two types of activities. The first, *(a,p).P*, executes the instantaneous event *(a,p)* and proceeds to *P*. The second, *A:P*, executes a resource-consuming action *A* during the first time unit and proceeds to process *P*. An action can take place if none of the resources used by it fails and assuming that it does not violate the system's power constraints. Otherwise, process *A:P* cannot execute the action and behaves as *NIL*. As a shorthand notation, we will write $A^n{:}P$ for a process that performs *n* consecutive actions *A* and then behaves as *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. In *P\F*, where *F* is a set of labels, the scope of labels in *F* is restricted to process *P*: components of *P* may use these labels to interact with one another but not with *P*'s environment. The construct $[P]_I$, $I \subseteq R$, produces a process that reserves the use of resources in *I* for itself. Finally, the process *rec X. P* denotes standard recursion.

We will explain P^{2}ACSR through some simple examples. Consider a simple system C (Figure 2.1.(a)), consisting of an input channel *in* and an output channel *out*, and a resource *cpu*, such that, every time a message is received via *in* the resource *cpu* is consumed for a single time unit and then a message is sent via channel out.



(a)

(b)

$$C \stackrel{def}{=} (in,1).\,C' + \varnothing{:}\,C$$
$$C' \stackrel{def}{=} \{(cpu,1,2)\} : (\overline{out},1).\,C$$

Figure 2.1. (a) Diagrammatic and (b) P^{2}ACSR description of system C.

The P^{2}ACSR description (Figure 2.1.(b)) specifies that, in its initial state, *C* may receive an input via channel *in* and then behave as *C'*. Alternatively, process *C* may idle for one time unit. On the other hand, process *C'* employs resource *cpu* at priority level *1* and power consumption level *2* to produce an output at channel *out*, and then returns to the initial state *C*.

Now consider the possibility that in any time unit, resource *cpu* may fail with probability 0.01, that is, $\pi(cpu) = 0.99$. If such a failure takes place, action *{(cpu,1,2)}* cannot take place and process *C'*, requiring the use of a failed resource, becomes inactive and deadlocks. Then, we may define a fault-tolerant version *FC* of system *C* as shown in Figure 2.2(a). Here, in case there is a failure of resource *cpu* , i.e., $\overline{cpu}$ is up, only one unit of power is consumed by the cpu and the process returns to the initial state without producing an output.

(a)

$$FC \overset{def}{=} (in,1).\ FC' + \varnothing : FC$$

$$FC' \overset{def}{=} \{(cpu,1,2)\} : (\overline{out},1).\ FC + \{(\overline{cpu},1,1)\} : FC$$

(b)

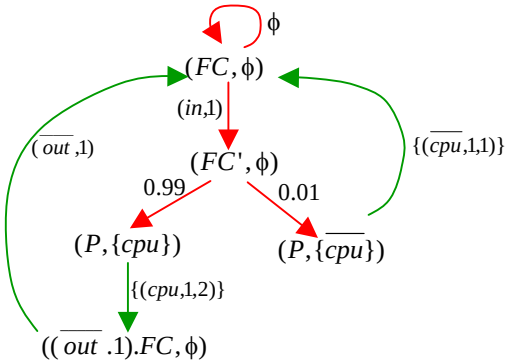


Figure 2.2. (a) P^{2}ACSR description of system FC and (b) its transition system.

Semantics. The informal account of behavior given of processes can be 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 transition system that captures the *nondeterministic* and *probabilistic* behavior of processes. Next, this transition system is refined into the *prioritized* transition system which implements priority arbitration between actions. The precise semantics rules are omitted but can be found in [10].

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*, the set of possible worlds involving *Z* is the set of all possible combinations of the resources in *Z* being up or down. Given a world *W* of a set of resources *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)*, representing a P^{2}ACSR 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.

The intuition for the semantics is as follows: for a P^{2}ACSR process *P*, we begin with the configuration *(P,∅)*. 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 P^{2}ACSR assumes that 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 illustrate the rules of the semantics with an example. Consider the process *FC* in its initial state *(FC,∅)*. This configuration has no immediately relevant resources. Hence, it can initially engage into one of two nondeterministic transitions, one labeled by the event *(in,1)* leading to state *(FC',∅)* and an idling action, labeled by ∅ leading back to itself:

$$(FC,\phi) \xrightarrow{(in,1)} (FC',\phi\}) \qquad (FC,\phi) \xrightarrow{\phi} (FC,\phi)$$

We continue with configuration *(FC',∅)*. Its immediately relevant resources are {*cpu*}. Since no information is contained in the configuration's world about *cpu*, we make two probabilistic transitions that determine the possible worlds of the resource, that is {*cpu*} and { $\overline{cpu}$ }.

$$(FC',\phi) \xrightarrow{\pi(cpu)}{}_p(FC',\{cpu\}) \qquad (FC',\phi) \xrightarrow{\pi(\overline{cpu})}{}_p(FC',\{\overline{cpu}\})$$

The resulting configurations allow one (initial) transition each

$$(FC',\{cpu) \xrightarrow{\{(cpu,1,2)\}} ((\overline{out},1).FC,\phi) \qquad (FC',\{\overline{cpu}\}) \xrightarrow{\{(\overline{cpu},1,1)\}} (FC,\phi)$$

and finally

$$((\overline{out},1).FC,\phi) \xrightarrow{(\overline{out},1)} (FC,\phi)$$

The complete transition system, for $\pi(cpu)=0.99$, can be seen in Figure 2.2.(b).

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. Examples of the preemption relation, whose precise definition can be found in [2], include the following:

1. $\{(r_1,1,4),(r_2,2,2)\} \prec \{(r_1,2,3),(r_2,4,1)\}$: a timed action preempts another timed action if it uses the same resources at a higher priority.
2. $(a,1) \prec (a,4)$: an instantaneous event preempts any other event with the same label and lower priority.
3. $\{(r_1,1,4),(r_2,2,2)\} \prec (\tau,1)$: an event labeled with τ and a nonzero priority preempts any timed action.

We define the prioritized transition system "$\rightarrow_n$", which refines "$\rightarrow$", to account for preemption as follows: $C \xrightarrow{\alpha}_n C'$ if and only if $C \xrightarrow{\alpha} C'$ is an unprioritized transition, and there is no unprioritized transition $C \xrightarrow{\beta} C'$ with $\alpha \prec \beta$.

EDF Scheduler Example. The next example comes from the area of schedulability analysis. The example describes a set of periodic tasks scheduled according to the Earliest Deadline First (EDF) [6] scheduling policy. This policy assigns dynamic priorities to the tasks based on how close they are to their deadlines. Each task T_i has a period p_i, a worst-case execution time c_i, and a deadline d_i by which the execution must be completed. Deadlines for all tasks are assumed to be equal to their periods.

A task set is modeled as a collection of processes $T_1,\ldots,T_n$, one process for each task. All tasks share the same processor, modeled by the resource *cpu*. No other tasks use the processor. A task is represented as the process T_i, shown below, and represented pictorially in Figure 2.3.

$$T_i = [(Job_i \mid Actuator_i) \setminus \{start_i\}]_{\{cpu,cont\}} \qquad i=\{1,\ldots,n\}$$

$$Actuator_i = (\overline{start_i},i).\varnothing^{p_i} : Actuator_i \qquad i=\{1,\ldots,n\}$$

$$Job_i = \varnothing : Job_i + (start_i,0).Exec_{i,0,0} \qquad i=\{1,\ldots,n\}$$

$$\begin{aligned} Exec_{i,e,t} = e < c_i \rightarrow (&\{(cpu, p_{\max} - (p_i - t),0),(cont,1,0)\} : Exec_{i,e+1,t+1} \\ &+ \{(cpu, p_{\max} - (p_i - t),0),(\overline{cont},1,0)\} : Job_i \\ &+ \varnothing : Exec_{i,e,t+1}) \end{aligned} \qquad i=\{1,\ldots,n\}$$

$$+ e = c_i \rightarrow Job_i \qquad i=\{1,\ldots n\}, e=\{0,\ldots c_i\}, t=\{0,\ldots,p_i\}$$

Each T_i is a parallel composition of two processes: Job_i and $Activator_i$. The role of the activator is to keep track of the timing constraint of the task. At the beginning of every period, $Activator_i$ sends the signal $start_i$ to Job_i, releasing the task for execution. It then idles for the period duration p_i and repeats the cycle. If the task has not finished its execution by the end of the period, it will not be able accept the next $start_i$ signal, resulting in a deadlock that will signify a scheduling failure. This is because in a parallel composition of processes a timed action can only occur if all processes can engage in a timed action. However, here, $Actuator_i$ can only send signal $start_i$, whereas $Exec_{i,e,t}$ can only engage in a timed action, thus bringing the process to a deadlock.

Upon receiving the $start_i$ signal, the other process, Job_i, begins the execution of the task. The execution process has two additional parameters: *e* is the accumulated execution time and *t* is the elapsed time. At each time step, the task has a priority that is increased as the task approaches its deadline. The task that has

been released *t* time units ago has p_i - *t* time units remaining until the deadline, and has priority $p_{max} - (p_i - t)$, where $p_{max}=max(p_1,\ldots,p_n)+1$. When the task receives the processor resource, it executes for one time unit and its accumulated execution time *e*, is increased together with the elapsed time *t*. At any time step, the task can be interrupted by another task that has a closer deadline. In this case, the task makes an idling step and its accumulated execution time stays the same while the elapsed time is increased.

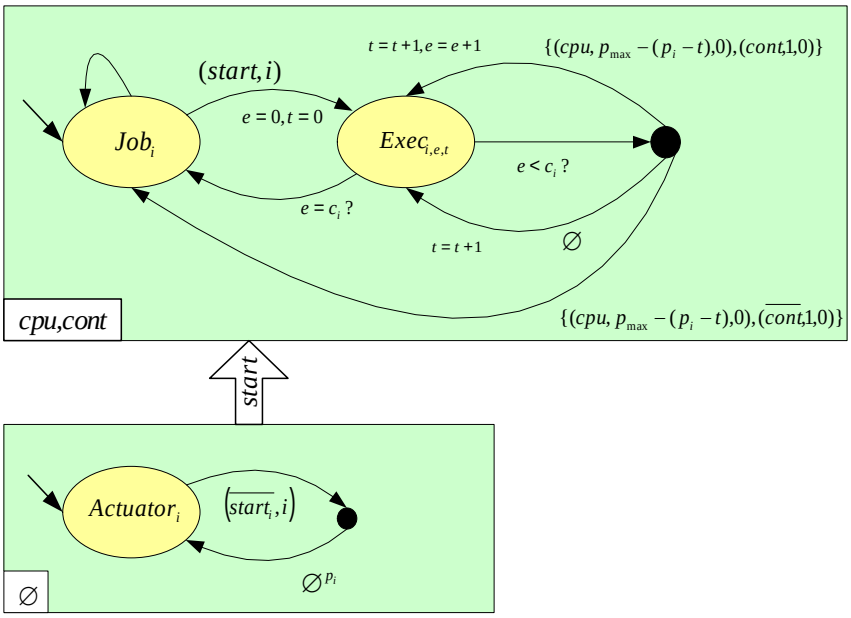


Figure 2.3. A model of an EDF task set

In order to model variable task-execution time, we introduce a probability distribution on the time it takes to complete the task. For simplicity, we assume that the execution time of a task conforms to the geometric distribution. That is, after every execution step, the task may complete with probability π and continue its execution with probability $1-\pi$. Thus, the probability that the task takes *i* time units to execute is $(1-\pi)^{i-1}\cdot\pi$. We assume that this distribution is the same for all tasks. We introduce an additional resource *cont* that represents the continuation of task execution. When the resource fails, the task completes its execution and becomes Job_i. Otherwise, the execution continues, up to the worst-case execution time.

The well-known results from [6] state that a set of tasks is schedulable if the utilization of the task set, $U_i = \Sigma_{i\varepsilon\{1..n\}} c_i/p_i$, does not exceed 1. The task set from our example satisfies this criterion, and, by checking the resulting process for the absence of deadlocks, we can indeed verify that all deadlines are met. Our method can be used to do the schedulability analysis of tasks that can block each other due to synchronization under EDF as well other scheduling algorithms.

3. Analysis

We can perform various kinds of analysis of power-aware real-time systems within the P^2ACSR formalism. As already mentioned in the example of the previous section, we can perform *schedulability analysis* of P^2ACSR processes to determine whether or not a real-time system with a particular scheduling discipline misses any of its deadlines. This kind of analysis can be carried out using *deadlock-detection* and/or *equivalence checking* of P^2ACSR processes. Deadlock detection can be performed by traversing the transition system of the process and looking for deadlocked states. On the other hand, equivalence checking is a verification technique aimed at deciding whether one system *implements* another with respect to some

notion of implementation relation. Among various implementation relations, strong and weak bisimulations have been defined and algorithms are given for their automatic verification (see [11] for example). It has been shown that bisimulations are useful in the schedulability analysis of real-time systems [7].

Another technique for analyzing P^{2}ACSR properties of specifications is *model checking*. Model checking is used to determine whether a system specification satisfies a property typically expressed as a temporal logic formula. To allow model checking on P^{2}ACSR specifications, we have introduced a probabilistic temporal logic that allows one to express power consumption constraints. The logic allows us to express, for example, that the power consumption of a system executing a given sequence of events never exceeds some threshold, or that probability of exceeding the threshold is small. The logic and its model-checking algorithm are presented in [10].

An additional useful metric for evaluating the power-consumption behavior of a process is that of *bounds of power consumption* over a fixed interval of time. It is possible to compute such bounds (the minimum and maximum power consumption during a time period) by traversing and analyzing the labeled transition system (i.e., labeled concurrent Markov chain) of a process, in time polynomial to the number of states of the transition system. Details can be found in [10].

Example. Consider the two systems in Figure 3.1. Each system repeatedly accesses a resource and signals success after each use. The system P employs a highly reliable resource r that never fails, $\pi(r)=1$, but requires a large amount of power to use it. On the other hand, the system Q uses a less reliable resource r' with $\pi(r') = ½$ that consumes less power.

(a)

$$P \overset{def}{=} \{(r,1,2)\}: (\overline{succ},1).P$$
$$Q \overset{def}{=} \{(r',1,1)\} : (\overline{succ},1).Q + \{(\overline{r}',1,0)\} : (\overline{succ},1).\ Q$$

(b)

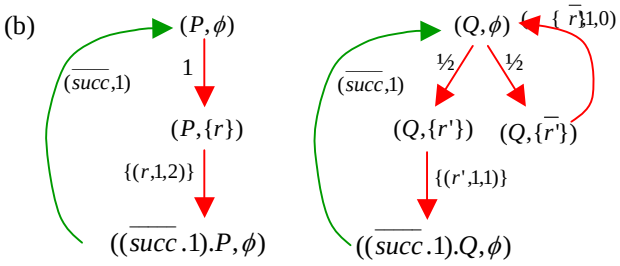


Figure 3.1. (a) P^{2}ACSR description of systems P and Q and (b) their transition systems.

Analyzing the transition systems, we see that, although Q risks a delay in successfully using resource r', it consumes less power on average than P per successful resource use. Using our temporal logic, we check the property that the system eventually performs a $\overline{succ}$ event with probability 1 and consumes at most 1 unit of power by that time. This property is satisfied by Q but not by P. On the other hand, the property that the system can perform a $\overline{succ}$ event within 2 time units with probability ¾, while consuming 2 units of power, is satisfied by both P and Q. Finally, if the probability is increased from ¾ to 1, the property is satisfied by P but not by Q. □

4. Power-Aware Real-Time Scheduling

In this section, we illustrate the use of P^{2}ACSR in the modeling and analysis of a power-aware application. The algorithm of power-aware scheduling is taken from [1] and uses dynamic voltage scaling [5] to optimize power consumption in an embedded real-time system. Dynamic voltage scaling allows us to make a tradeoff between performance and power consumption. A CMOS-based processor operating on a

lower frequency can use a lower supply voltage and thus consume less power. At the same time, a lower-frequency execution means that tasks take longer to compute. If the system has real-time requirements associated with it, these requirements may become violated at lower frequency. A power-aware real-time operating system has to decide when it is possible to operate at a lower frequency while at the same time maintaining the timing properties of the system.

Pillai and Shin propose extensions to real-time scheduling algorithms to make use of dynamic voltage scaling [1]. We use their approach to extend the model of an EDF task set presented in Section 2 to utilize cycles unused by the tasks to lower the operating frequency for other tasks. The ratios of execution time to period in each task define the nominal utilization of the processor by the task set that determines whether the tasks can be scheduled. In reality, tasks often take much less than the worst case to execute. Thus, the effective utilization of the task set may be much lower than the nominal one. When the processor operates at a lower frequency, execution times of tasks grow accordingly. This may increase the nominal utilization so much that the task set may be considered no longer schedulable. However, the effective utilization may still be small enough even at a lower frequency. The power-aware scheduling algorithm of [1] computes effective utilization during execution and switches frequencies to use the lowest frequency for which the task set remains effectively schedulable.

The algorithm of [1] recomputes the operating frequency every time a task is released for execution or ends its execution in the current period based on the effective utilization of each task. Initially, and upon release of a task, its effective utilization is set to the largest value, $U_i = c_i / p_i$. When the task ends a period, U_i is set to the actual utilization c^{act} / p_i. Then, it selects the least operating frequency f_i for the processor such that $U_1 + \ldots + U_n \leq f_i / f_m$, where f_m is the largest possible operating frequency.

We now show how to represent this algorithm as a P^{2}ACSR process. We first extend the model of a task with the ability to execute faster or slower depending on the state of the system. The task is similar to the one shown in Section 2, except that one execution can take one time unit or two time units depending on the operating frequency of the processor. The task uses events *fast* and *slow* to determine whether the processor is in the fast or slow mode. If the processor is in the slow mode, the next computation step takes two time units. The task T_i uses two additional events, $release_i$ and $end_{i,j}$. These events are used to drive the voltage-scaling algorithm and correspond to the release of task T_i and the completion of T_i after j time units, respectively.

Resources used in the model of the task, *cpu* and *cont*, do not consume power since both of them represent abstract notions: scheduling priorities and probabilistic completion. Power consumed by the processor is captured by a separate resource *power* that is used by the process *DVS* described below. The process *DVS*, shown in Figure 4.1, consists of two parallel components. One subprocess, represented by the process $Scale_{e_1,e_2,e_3}$, represents the voltage-scaling algorithm itself. Triggered by an event $release_i$ or $end_{i,j}$ that corresponds to the release or completion, respectively, of the task T_i after executing for j time units, the process *SetNew* computes the effective utilization. It sends the signal f_{down} if a lower operating frequency is possible and the signal f_{up} otherwise. The other component of the process *DVS* keeps the information of the current operating frequency. It has two states, DVS_{fast} and DVS_{slow}. In the former state, the process uses the resource *power* at the power consumption level of pw_{fast}, and in the latter state, it uses the same resource with power consumption of pw_{slow}, where pw_{fast} and pw_{slow} are parameters of the model. Note that the priority of events f_{up} and f_{down} is greater than the priority of events *fast* and *slow*. This ensures that the tasks always receive the latest status of the processor.

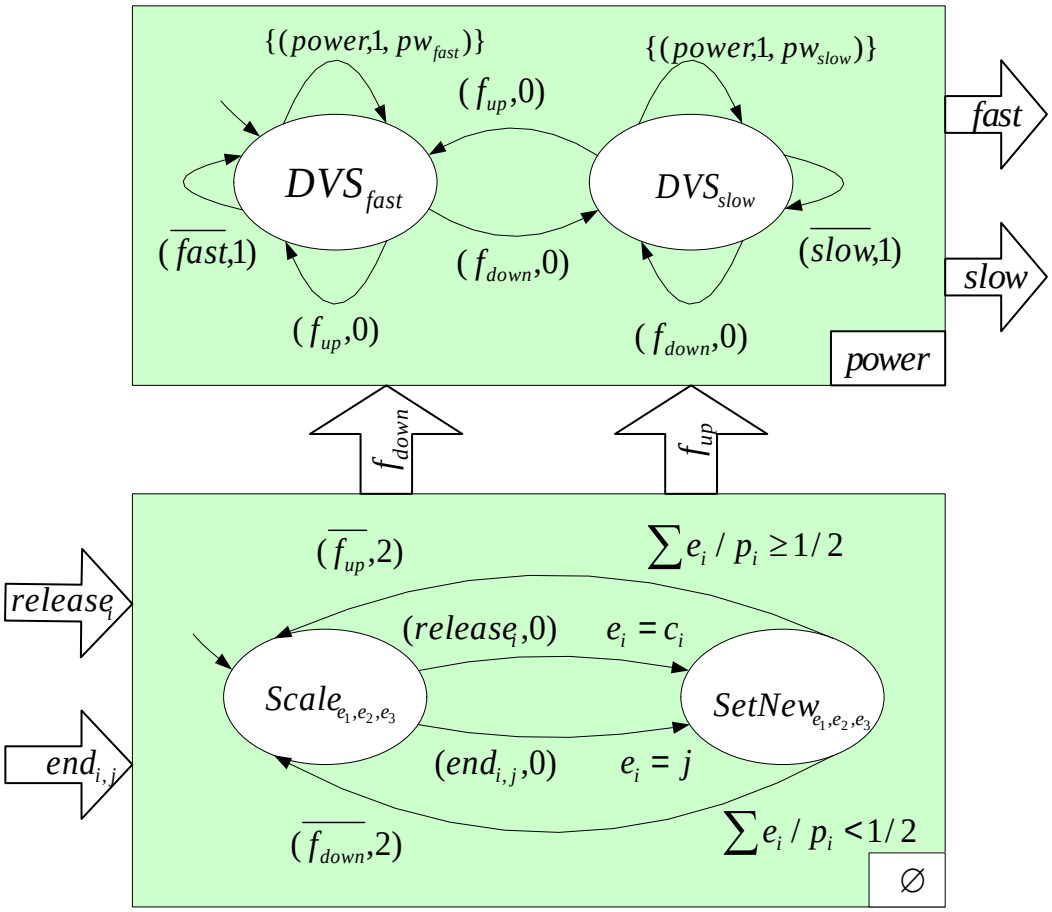


Figure 4.1. P^{2}ACR representation of voltage scaling

Analysis. We explored the set of three tasks with parameters shown in Table 1. First, we checked that the task set is schedulable by the power-aware scheduling algorithm. The resulting system does not have any deadlocks, which means that all timing constraints are satisfied. We also calculated the expected power consumption of the task set for the duration of one major frame, that is, the product of periods of all tasks, $p_1 \cdot p_2 \cdot p_3$, 1120 time units. The probability of the task completion after a computation step was taken to be 1/3, and parameters pw_{fast} and pw_{slow} were 2 and 1, respectively. The expected minimum and maximum power consumption was calculated to be 1906.66 and 1922.65, respectively. Without the dynamic voltage scaling, when each step would take pw_{fast} power units, the power consumption for the same period would be 2240 units. As a result, expected savings from the dynamic voltage scaling are between 14% and 14.8%.

Task	Execution time	Period
1	3	8
2	3	10
3	1	14

Table 1. Example task set

5. Conclusions

Formal modeling and analysis of power-aware real-time systems offer important advantages in the design of embedded systems. We have presented P^2ACSR, process algebra for resource-oriented real-time systems. The formalism allows one to model the power consumption of resources and perform power-oriented analysis of a system's behavior. Since tool support is critical for the success of formal techniques, we are extending the PARAGON toolset [8] allows the specification and analysis of P^2ACSR processes.
We have implemented deadlock detection and equivalence checking of processes, as well as calculation of the probabilistic bounds on power consumption. We plan to implement the model-checking algorithm in near future.

One useful measure to be computed on P^2ACSR specifications is that of long-run average performance. Average behavior is particularly interesting for power consumption studies. Average power consumption can be computed per unit of time or per period of time. It has already been shown in the literature how to evaluate the long-run average behavior of a probabilistic system [9]. As future work, we intend to study the adaptation of the above-mentioned technique to P^2ACSR and implement it in the PARAGON toolset.

References

1. P. Pillai and K. G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, Proceedings of the 18th Annual ACM Symposium on Operating Systems Principles, 2001.
2. 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*, Jan 1994, pp. 158-171.
3. A. Philippou, O. Sokolsky, R. Cleaveland, I. Lee, and S. Smolka, Probabilistic resource failure in real-time process algebra, Proceedings of CONCUR '98, August 1998, pp. 389-404.
4. M. Vardi, Automatic verification of probabilistic concurrent finite-state programs, Proceedings of the 26th Annual Symposium on Foundations of Computer Science, 1985, pp. 327—338.
5. T. D. Burd and R. W. Brodersen, Energy efficient CMOS microprocessor design, Proceedings of the 28th Annual Hawaii International Conference on System Sciences. Volume 1: Architecture, pp. 288-297, IEEE Computer Society Press, 1995.
6. C. L. Liu and J. W. Layland, Scheduling algorithms for multiprogramming in a hard real-time environment, *Journal of the ACM*, 20(1), pp. 46-61, 1973.
7. J-Y. Choi and I. Lee and H-L Xie, The Specification and Schedulability Analysis of Real-Time Systems using ACSR. In Proc. of the Real-Time Systems Symposium. IEEE Computer Society Press, 1995.
8. O. Sokolsky, I. Lee, and H. Ben-Abdallah, Specification and Analysis of Real-Time Systems with PARAGON, *Annals of Software Engineering*, 7:211-234, 1999.
9. L. de Alfaro, How to Specify and Verify the Long-Run Average Behavior of Probabilistic Systems, Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science, pp. 454-465. IEEE Computer Society Press, 1998.
10. I. Lee, A. Philippou and O. Sokolsky, Formal Mdodeling and Analysis of Power-Aware Real-Time Systems. Technical Report, MIS-CIS-02-12, Department of Computer and Information Science, University of Pennsylvania, 2002.
11. A. Philippou, O. Sokolsky and I. Lee, Weak Bisimulation for Probabilistic Systems. In Proc. of CONCUR'00, pages 334-339. LNCS 1877, Springer Verlag, 2000.

Insup Lee and Oleg Sokolsky are with the Department of Computer and Information Science, University of Pennsylvania, USA, {lee,sokolsky}@cis.upenn.edu. Anna Philippou is with the Department of Computer Science, University of Cyprus, Cyprus, annap@ucy.ac.cy.

This research was supported in part by ARO DAAD19-01-1-0473, NSF CCR-9988409, NSF CCR-0086147, NSF CISE-9703220, and ONR N00014-97-1-0505.