Modeling and Analysis of Power-Aware Systems*

Oleg Sokolsky[1], Anna Philippou[2], Insup Lee[1], and Kyriakos Christou[1]

[1] University of Pennsylvania, USA. {sokolsky,lee,christou}@cis.upenn.edu
[2] University of Cyprus, Cyprus. annap@ucy.ac.cy

Abstract. The paper describes a formal approach for designing and reasoning about power-constrained, timed systems. The framework is based on *process algebra*, a formalism that 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 model alternative power-consumption behaviors and analyze tradeoffs in their timing and other characteristics. This paper describes the modeling and analysis techniques, and illustrates them with examples, including a dynamic voltage-scaling algorithm.

1 Introduction

In recent years, great technological advances in wireless communication and mobile computing have given rise to sophisticated embedded devices (e.g., PDA, cell phones, smart sensors) and wireless network infrastructures that are becoming widespreadly 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 timing constraints and other undesirable properties. A challenge presented by such systems is the development of algorithms that incorporate power-saving techniques and task management without sacrificing timing and performance guarantees, see e.g. [14].

The main purpose of this paper is to present a unified formal framework and associated toolset for designing and reasoning about power-constrained, timed

* This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CCR-0209024, ARO DAAD19-01-1-0473, and by the EU Future and Emerging Technologies programme IST-1999-14186 (ALCOM-FT).

H. Garavel and J. Hatcliff (Eds.): TACAS 2003, LNCS 2619, pp. 409–424, 2003.

systems. The framework we propose is based on *process algebra*, a formalism which has been developed to describe and analyze communicating, concurrent systems. The most salient aspect of process algebras is that they support the *modular* specification and verification of systems. Process algebras are being used widely in specifying and verifying concurrent systems and they have been extended to account for time and probabilistic behavior.

The formal framework we propose is based on the process algebra P^{2}ACSR which extends our previous work on formal methods for real-time [10] and probabilistic systems [13] by incorporating the ability of reasoning about power consumption. The Algebra of Communicating Shared Resources (ACSR) [10] is a timed process algebra which represents a real-time system as a collection of concurrent processes. Each process can engage in two kinds of activities: communication with other processes by means of instantaneous *events* and computation by means of timed *actions*. Executing an action requires access to a set of resources and takes a non-zero amount of time measured by an implicit global clock. The notion of a resource, which is already important in the specification of real-time systems, additionally provides a convenient abstraction for a variety of aspects of systems behavior. One such aspect is the failure of physical devices: in a probabilistic extension of ACSR, PACSR [13], resources are extended with the ability to fail, and are associated with a probability of failure. In P^{2}ACSR, the resource model of PACSR is further extended to reason about power consumption. Resources in P^{2}ACSR specifications are accompanied with information about the power consumption of each resource use. Thus, we can compute the power consumed by system executions requiring access to a set of power-consuming resources. We provide an operational semantics of P^{2}ACSR via *labeled concurrent Markov chains* [17], which are transition systems containing both probabilistic and nondeterministic behavior. Probabilistic behavior is present in the model due to resource failure and nondeterministic behavior due to the fact that P^{2}ACSR specifications typically consist of several parallel processes producing events concurrently.

We are interested in being able to specify and verify high-level requirements of P^{2}ACSR specifications, and, in particular, to study their power-consumption behavior and tradeoffs in their timing and other characteristics. To do this we extend model-checking analysis techniques to allow reasoning about power consumption properties. First, we present a probabilistic, power-aware temporal logic for expressing properties of P^{2}ACSR expressions by associating power-consumption constraints with temporal operators. Second, the analysis technique allows us to compute bounds on power consumption in executions of the model. We present model-checking algorithms both for the logic and for bound computations. In order to allow the automatic analysis of P^{2}ACSR specifications, we have extended the PARAGON toolset [16], which previously allowed the specification and analysis of ACSR and PACSR processes, to accept P^{2}ACSR specifications and construct the probabilistic transition systems emanating from them, and we have augmented the toolset with the model-checking algorithms both for the logic and for the bounds computations.

We illustrate the usefulness of the proposed formalism using a dynamic voltage-scaling algorithm for real-time, power-aware systems [14]. In this example, we use resources to model the power-consuming processing unit which can be used at different power levels with different execution speeds. We model the probabilistic nature of task execution time in the system by employing probabilistic resources. The expected savings in power consumption that the algorithm offers were computed automatically using the PARAGON toolset extension.

Related work. This work extends our earlier work on probabilistic process algebra PACSR [13]. Similar approaches process models have been considered by [8] and [15]. Similar logics and model checking approaches were presented in [15] and [8]. Extensions made in this work allow us to reason quantitatively about power consumption using resource attributes as rewards. Much work on reward-based modeling and analysis has been done in the context of performance modeling using the formalisms of stochastic process algebras [2] and continuous-time Markov chains [1]. The approach taken in this paper uses discrete time and yields coarser but easier to analyze models. A difference in the modeling approach, compared to all these papers, is that we use resource attributes to capture probabilistic aspects of behavior as well as rewards, which offers us a flexible and easily extensible framework.

The rest of the paper is organized as follows: the next section present the P^{2}ACSR syntax and semantics. Section 3 describes analysis techniques for P^{2}ACSR 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 future work.

2 The General Framework

We assume that a system contains a finite set of serially reusable resources drawn from a countable infinite set of resources $\mathcal{R}$ and we let r range over $\mathcal{R}$. Resources correspond to physical entities, such as processor units and communication channels, or to abstract notions such as message arrival. They can have a set of numerical attributes that let us capture quantitative aspects of resource-constrained computations. We use three resource attributes. One attribute captures the probability of resource failure, which remains constant for each resource throughout a system specification. The other two attributes may change with each resource use and represent access priority and power consumption.

Probabilistic resource failures. We associate with each resource a probability [13]. This probability captures the rate at which the resource may fail. A failure may correspond to either a physical failure, such as a processor failure, or a failure of some abstract condition, for example no message arrival when one was expected. We assume that in each execution step, resources fail independently. To capture the notion of a failed resource we also consider the set $\overline{\mathcal{R}}$ that contains, for each $r \in \mathcal{R}$, its dual element $\overline{r}$, representing the *failed* resource r. For each $r \in \mathcal{R}$, we use $\pi(r) \in [0, 1]$ to denote the probability of resource r being

up in a given step, while $\pi(\overline{r}) = 1 - \pi(r)$ is the probability of r failing in a given step. Failed resources are useful when we need to model recovery from failures.
Resources and power consumption. In order to reason about power consumption in distributed settings, the set of resources $\mathcal{R}$ is partitioned into a finite set of disjoint classes $\mathcal{R}_i$, for some index set I. Intuitively, each $\mathcal{R}_i$ corresponds to a distinct power source which can provide a limited amount of power at any given time. This limit is denoted by c_i. Each resource $r \in \mathcal{R}_i$ consumes a certain amount of power from the source $\mathcal{R}_i$. As we will see below, the rate of power consumption is specified in *timed actions.*

2.1 The Syntax

As PACSR, P^{2}ACSR has three types of actions: instantaneous events, timed actions, and probabilistic actions. We discuss these three concepts below.
Instantaneous events. Instantaneous actions are called *events*, and provide the basic synchronization primitives in the process algebra. An event is denoted as a pair (a, p), where a is the label of the event and p, a natural number, is the priority of the event. Labels are drawn from the set $\mathsf{L} = \mathcal{L} \cup \overline{\mathcal{L}} \cup \{\tau\}$, where if $a \in \mathcal{L}$, $\overline{a} \in \overline{\mathcal{L}}$ is its *inverse* label. The special label τ, also called the silent action, arises when two events with inverse labels are executed concurrently. Thus, events are similar to actions in CCS, with the distinction that here we also impose priorities. We use $\mathcal{D}_E$ to denote the set of events.
Timed actions. A timed action consists of several resources, each resource being used at some priority and at some level of power consumption, and consumes one unit of time. Formally, an action is a finite set of triples of the form (r, p, c), where r is a resource, p is the priority of the resource usage and c is the rate of power consumption, with the restriction that each resource is represented at most once.

An example of an action is given by $\{(cpu, 2, 3), (msg, 1, 0)\}$. This action takes one unit of time and uses resource *cpu* representing a processor unit, at priority level two, consuming three units of power. This action also assumes that the processor receives a message, represented by resource *msg*. The fact that the processor may fail or that the message may or may not arrive is modeled by assigning probabilities of failures to resources *cpu* and *msg*. The action takes place only if none of the resources *cpu* and *msg* fail. On the other hand, action $\{(cpu, 2, 3), (\overline{msg}, 1, 0)\}$ takes place when resource *msg* fails and resource *cpu* does not. Such an action may be used to describe the behavior of the processor when it does not receive a message. The action $\emptyset$ represents idling for one unit of time, since no resource is consumed. We denote the set of resources used in an action A as $\rho(A)$. We use $\mathcal{D}_R$ to denote the set of actions.
Probabilistic transitions. As already mentioned resources are associated with a probability of failure. Thus, the behavior of a resource-consuming system has certain probabilistic aspects to it. Consider the action $\{(cpu, 2, 3), (msg, 1, 0)\}$, where resources *cpu* and *msg* have probabilities of failure 0 and 1/3, respectively, that is $\pi(cpu) = 1$ and $\pi(msg) = 2/3$. This action takes place with probability $\pi(cpu) \cdot \pi(msg) = 2/3$ and fails with probability 1/3.

Processes. We let P, Q range over processes and we assume a set of process constants, each with an associated definition of the kind $X \overset{\text{def}}{=} P$. We write Proc for the set of P^{2}ACSR processes. The following grammar describes the syntax of P^{2}ACSR processes. We present only those operators that are used in the examples in the paper. The complete set of operators can be found in [11].

$$P ::= \text{ NIL} \mid (a, n).\, P \mid A{:}P \mid b \to P \mid P + P \mid P \| P \ \mid P \backslash F \mid X$$

Process NIL represents the inactive process. There are two prefix operators, corresponding to the two types of actions. The first, $(a, n).\, P$, executes the instantaneous event (a, n) and proceeds to P. When it is not relevant for the discussion we omit the priority of an event in a process and simply write $a.\, P$. The second, $A : P$, executes a resource-consuming action A during the first time unit and proceeds to P. An action can take place if none of the resources used by it fail and also if it does not violate the power constraints of the system. Otherwise, $A : P$ 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. Process $b \to P$ behaves as P if condition b is true, otherwise it behaves as NIL. Process $P + Q$ represents a nondeterministic choice between the two summands. 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 $P \backslash F$, where $F \subseteq L$, the scope of labels in F is restricted to process P; that is, components of P may use these labels to interact with one another but not with P's environment.

As an example of a process, consider the process

$$P \overset{\text{def}}{=} \{(cpu, 2, 3), (msg, 1, 0)\} : P_1 + \{(cpu, 2, 2), (\overline{msg}, 1, 0)\} : P_2 \ .$$

Process P represents a processor that can accept messages from a channel. Depending on whether the message arrives or not, P has two alternative behaviors. If the message arrives, that is, resource msg is up, P processor receives the message, consuming 3 units of power, and proceeds to process it as P_1. Otherwise, $\overline{msg}$ is up, P consumes only 2 units of power and continues as P_2.

As a syntactic convenience, we allow P^{2}ACSR processes to be parameterized by a set of index variables, allowing us to represent collections of similar processes concisely. Each index variable is given a fixed range of values. For example, the parameterized process $P_t \overset{\text{def}}{=} t < 2 \to (a_t, p_t).P_{t+1}, \quad t \in \{0..2\}$ is equivalent to the following three processes: $P_0 \overset{\text{def}}{=} (a_0, p_0).P_1, \quad P_1 \overset{\text{def}}{=} (a_1, p_1).P_2, \quad P_2 \overset{\text{def}}{=}$ NIL.

2.2 Operational Semantics

In this section we give an informal account of the operational semantics for P^{2}ACSR. The detailed account can be found in [11] and is an extension of the PACSR semantics presented in [13]. The behavior of a P^{2}ACSR process depends on the status of the resources it requires during its first time step. For example, the process P in the previous section, will evolve depending of whether the processor is available or failed and whether or not a message arrives. In order to capture this relation between process behavior and resource status we introduce the notion of a *world.* A *world* is a set of resources W such that it cannot contain both a resource and its failed counterpart. When $r \in W$, r is known to be available, when $\overline{r} \in W$, r is known to be failed. Then, we introduce the notion of a *configuration* as a pair of the form $(P, W) \in \mathsf{Proc} \times 2^{\mathsf{R}}$, representing a P^{2}ACSR process P in world W. We write S for the set of all configurations. Further, given world W we write $\mathcal{W}(W)$ for the set of worlds that give status to the same resources as W, and $\pi(W)$ is the probability of the world, given by the product of $\pi(r), r \in W$.

The intuition for the semantics is as follows: for a process P, we begin with the configuration $(P, \emptyset)$. As computation proceeds two types of transitions may be performed: (1) for any configuration (P, W) where the world W does not contain information regarding the status of *all* of P's immediately relevant resources, probabilistic transitions are taken to a number of new configurations each of which spells out a possible world of these resources. Such configurations are called probabilistic and denoted S_p. Otherwise, (2) for all configurations holding all necessary information about the status of resources, nondeterministic transitions (which can involve events or actions) are taken. Such configurations are called nondeterministic and denoted S_n. The set of immediately relevant resources, denoted $\mathsf{imr}(P)$, is defined inductively on the structure of the process. Intuitively, immediately relevant resources are contributed by the timed actions that may be taken by the process in the first step and also resources added by the resource closure operator. After the status of a resource is determined by a 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.

Thus the semantics is given in terms of a labeled transition system whose states are configurations and whose transitions are either probabilistic or nondeterministic. Each probabilistic transition originates from configurations in S_p and leads to a configuration in S_n. Probabilistic transitions are labeled with the probability of reaching a new world with the updated resource status. The rule for the probabilistic transition relation describes the manipulation of the worlds as a result of the transition.

As an example, consider the process $P \overset{\text{def}}{=} \{(r_1, 2, 1), (r_2, 2, 2)\} : P_1 + (e, 1).P_2$ in the initial configuration $(P, \emptyset)$. The immediate resources of P are $\{r_1, r_2\}$. Since there is no knowledge in the configuration's world regarding these resources, the configuration belongs to the set of probabilistic configurations S_p, from where we have four probabilistic transitions that determine the status of r_1 and r_2:

$$(P, \emptyset) \xrightarrow{\pi(r_1)\cdot\pi(r_2)}_p (P, \{r_1, r_2\}),\ (P, \emptyset) \xrightarrow{\pi(r_1)\cdot\pi(\overline{r_2})}_p (P, \{r_1, \overline{r_2}\}),$$

$$(P, \emptyset) \xrightarrow{\pi(\overline{r_1})\cdot\pi(r_2)}_p (P, \{\overline{r_1}, r_2\}),\ (P, \emptyset) \xrightarrow{\pi(\overline{r_1})\cdot\pi(\overline{r_2})}_p (P, \{\overline{r_1}, \overline{r_2}\}).$$

All of the resulting configurations are nondeterministic since they contain full information about P's immediate resources.

Nondeterministic transitions are labeled with either an event or a timed action. The rules for nondeterministic transitions are, for the most part, the same as for PACSR and can be found in [13]. The difference comes in the side conditions in the rules for action prefix and parallel composition. An action can take place if it does not violate the power consumption constraints. The predicate $\mathsf{valid}(A) = \bigwedge_{i \in I}(\sum_{r \in \mathcal{R}_i} \mathsf{pc}_r(A) \leq c_i)$ captures this requirement. The action prefix rule requires that the action appearing in the action prefix be valid. The rule for parallel composition requires that processes in a parallel composition need to synchronize on a timed action, that is, a process advances only if both of its subprocesses can take action steps labeled by actions A_1 and A_2 that use disjoint resources, and the resulting action $A_1 \cup A_2$ is valid.

In the example, the nondeterministic configuration $(P, \{r_1, r_2\})$, where $P \overset{\text{def}}{=} \{(r_1, 2, 1), (r_2, 2, 2)\} : P_1 + (e, 1).P_2$ has two nondeterministic transitions:

$$(P, \{r_1, r_2\}) \xrightarrow{\{(r_1,2,1),(r_2,2,2)\}} (P_1, \emptyset) \qquad \text{and} \qquad (P, \{r_1, r_2\}) \xrightarrow{e} (P_2, \{r_1, r_2\}).$$

The other configurations, $(P, \{r_1, \overline{r_2}\})$, $(P, \{\overline{r_1}, r_2\})$, and $(P, \{\overline{r_1}, \overline{r_2}\})$, allow only the e-labeled transition since either r_1 or r_2 is failed and the action cannot occur.

The prioritized transition system for P^{2}ACSR is based on the notion of *preemption* and refines the unprioritized transition relation $\longrightarrow$ by taking priorities into account. It is given by the pair of transition relations $\longrightarrow_p$ and $\longrightarrow_n$, the latter of which is defined below. The preemption relation $\prec$ on *Act* is defined as for ACSR, specifying when two actions are comparable with respect to priorities. For example, $\emptyset \prec A$ for all actions A, that is, the idle action $\emptyset$ is preempted by all other timed actions, and $(a, p) \prec (a, p')$, whenever $p < p'$. For the precise definition of $\prec$ we refer to [10]. The basic idea behind $\longrightarrow_n$ is that a nondeterministic transition of the form $(P, W) \xrightarrow{\alpha} (P', W')$ is included in $\longrightarrow_n$ if and only if there are no higher-priority transitions enabled in (P, W). Thus, the prioritized nondeterministic transition system is obtained from the unprioritized one by pruning away preemptable transitions.

3 Analysis

In this section we discuss possible analysis that can be performed on P^{2}ACSR specifications. We begin by presenting the formal model underlying P^{2}ACSR processes which is that of *labeled concurrent Markov chains* [17].

Definition 1. A *labeled concurrent Markov chain* (LCMC) is a tuple $\langle S_n, S_p, Act, \longrightarrow_n, \longrightarrow_p, s_0 \rangle$, where S_n is the set of nondeterministic states, S_p is the set of probabilistic states, *Act* is the set of labels, $\longrightarrow_n \subset S_n \times Act \times (S_n \cup S_p)$ is the nondeterministic transition relation, $\longrightarrow_p \subset S_p \times [0, 1] \times S_n$ is the probabilistic transition relation, satisfying $\Sigma_{(s,\pi,t) \in \longrightarrow_p} \pi = 1$ for all $s \in S_p$, and $s_0 \in S_n \cup S_p$ is the initial state. □

We may see that the operational semantics of P^{2}ACSR yields transition systems that are LCMCs with $Act = \mathcal{D}_E \cup \mathcal{D}_R$, and the sets S_n, S_p are the sets of nondeterministic and probabilistic configurations, respectively. In what follows, we let ℓ range over $Act \cup [0, 1]$.

A *computation* in $T = \langle S_n, S_p, Act, \longrightarrow_n, \longrightarrow_p, s_0 \rangle$ is either a finite sequence $c = s_0\, \ell_1\, s_1 \ldots \ell_k\, s_k$, where s_k has no transitions, or an infinite sequence $c = s_0\, \ell_1\, s_1 \ldots \ell_k\, s_k \ldots$, such that $s_i \in S_n \cup S_p$, $\ell_{i+1} \in Act \cup [0, 1]$ and $(s_i, \ell_{i+1}, s_{i+1}) \in\ \longrightarrow_p \cup \longrightarrow_n$, for all $0 \leq i$. We denote by $\mathsf{comp}(T)$ the set of all computations of T and by $\mathsf{Pcomp}(T)$ the set of all partial computations of T, that is the set of initial subsegments of computations of T.

To define probability measures on computations of an LCMC the nondeterminism present must be resolved. To achieve this, the notion of a *scheduler* has been employed [17,8]. A scheduler σ is an entity that, given a partial computation ending in a nondeterministic state, chooses the next transition to be executed. This gives rise to computation trees that can be viewed as labeled Markov chains. Each path through a computation tree is a *scheduled computation* of the LCMC and can be assigned a probability by taking a product of the probabilistic labels along the path. See [11] for the details.

3.1 Model Checking for P^{2}ACSR

The first technique we propose for analyzing P^{2}ACSR specifications is that of model-checking. Model checking is a verification technique aimed at determining whether a system specification satisfies a property typically expressed as a temporal logic formula. To allow model checking on P^{2}ACSR specifications, in this section we introduce a probabilistic temporal logic that allows one to associate power consumption constraints with fragments of behaviors. Behavioral fragments of interest are expressed in terms of regular expressions over *Act*, the set of observable actions. The associated model-checking algorithm, also presented in this section, is used to check whether these constraints are satisfied and thus whether formulae of the logic are satisfied by system specifications.

Our logic for P^{2}ACSR is an extension of the logic of [13], which, in turn, is based on the Hennessy-Milner Logic (HML) with *until* [7]. The extension established allows for quantitative analysis of power consumption properties of a system by associating a condition with the *until* operator. The condition takes a form such as $\leq pc$ or $\geq pc$ for a constant pc. In this way we can express a property that an execution, timed or untimed, satisfies a power consumption constraint, with a certain probability (which may be equal to one). We also include a second construct that allows a similar type of reasoning but specifying the power sources for which analysis is to be performed.

Definition 2. *(Power-aware PHML with until)* The syntax of $\mathcal{L}^{pc}_{PHMLu}$ is defined by the following grammar, where f, f', range over $\mathcal{L}^{pc}_{PHMLu}$-formulae, Φ is a regular expression over *Act*, R is a subset of the set of resources $\mathcal{R}$, p a number in $[0, 1]$ representing a probability, t a number representing a time limit, pc a number representing a power consumption, and $\bowtie \in \{\leq, <, \geq, >\}$:

$$f ::= tt \mid \neg f \mid f \wedge f' \mid f \langle \Phi \rangle^{\bowtie' pc}_{\bowtie p} f' \mid f \langle \Phi \rangle^{\bowtie' pc}_{\bowtie p, t} f' \mid f \langle \Phi \rangle^{\bowtie' pc, R}_{\bowtie p} f' \mid f \langle \Phi \rangle^{\bowtie' pc, R}_{\bowtie p, t} f'. \ \square$$

$\mathcal{L}^{pc}_{PHMLu}$-formulae are interpreted over states of LCMCs. Informally, formulae of the form $f\langle\Phi\rangle f'$ state that there is some execution and some integer l such that f holds for the first $l-1$ steps and f' becomes true in the l-th step and the observable behavior of the l-step execution involves some behavior from Φ. The subscript $\bowtie p$ denotes that the probability of paths fulfilling the formula is $\bowtie p$ and the use of subscript t denotes that the paths of interest are only those that achieve the goal in at most t time units. Finally, the superscript $\bowtie' pc$ requires paths to use $\bowtie' pc$ units of power, and the use of R, restricts power consumption calculations to the set of resources R. For instance, formula $tt\langle Act^*\rangle^{\leq pc}_{\geq 1} f$ expresses that there is some execution of the system for which eventually f becomes true, with probability 1, without consuming more than pc units of power. Similarly, $\neg(tt\langle Act^*\rangle^{\geq pc}_{>0} tt)$, specifies that the power consumption never exceeds the threshold of pc units, whereas $\neg(tt\langle Act^*\rangle^{\geq pc,\{cpu\}}_{>0} tt)$, specifies that the power consumption of resource cpu never exceeds the threshold of pc units.

In order to present the semantics of the four until operators, we need to compute the probabilities that certain behaviors occur. Consider now the formula $f\langle\Phi\rangle^{\leq pc,R}_{\bowtie p,t} f'$. Given two sets of states A, B of an LCMC T and a sequence of actions $\Phi \subseteq Act^*$, we consider following set of partial computations of T. The computations lead to a state in B via Φ, with intermediate states in A, and take less than time t and consume no more than pc units of power on resources in R. Given a scheduler σ, the set of complete scheduled computations that are extensions of the partial computations above is measurable in the probability space of T. We denote its probability $\mathrm{Pr}_A(T, \Phi, B, \leq pc, R, t, \sigma)$. Similarly, we can define probabilities for other kinds of formulas.

Finally, the satisfaction relation $\models\ \subseteq (S_n \cup S_p) \times \mathcal{L}^{pc}_{PHMLu}$, stating when an LCMC state satisfies a given formula, is defined inductively as follows. Let $T = (S_n, S_p, Act, \longrightarrow_n, \longrightarrow_p, s_0)$, be an LCMC. Then:

$s \models tt$ always
$s \models \neg f$ *iff* $s \not\models f$
$s \models f \wedge f'$ *iff* $s \models f$ and $s \models f'$
$s \models f\,\langle\Phi\rangle^{\bowtie' pc,R}_{\bowtie p,t} f'$ *iff* there is $\sigma \in \mathsf{Sched}(s)$ such that $\mathrm{Pr}_A(s, \Phi, B, \bowtie' pc, R, t, \sigma) \bowtie p$,
where $A = \{s' \mid s' \models f\}$, $B = \{s' \mid s' \models f'\}$

Similar definitions are given for the other variants of the until operator.

The Model-Checking Algorithm. Let *closure*(f) denote the set of formulae $\{\ f', \neg f' \mid f'$ is a subformula of $f\ \}$. Our model-checking algorithm is similar to the CTL model-checking algorithm of [5]. In order to check that LCMC T satisfies some formula $f \in \mathcal{L}^{pc}_{PHMLu}$, the algorithm labels each state s of T with a set $F \subseteq$ *closure*(f), such that for every $f' \in F$, $s \models f'$. T satisfies f if and only if s_0, the initial state of T, is labeled with f. The algorithm starts with the atomic subformulae of f and proceeds to more complex subformulae. The labeling rules are straightforward from the semantics of the operators, with the exception of the *until* operator.

In order to decide whether a state s satisfies one of the four until operators, we compute the maximum or minimum probability of the specified behavior Φ.

The maximum value of $\mathrm{Pr}_A(s, \Phi, B, \leq pc, \sigma)$ over all σ is computed by solving a linear programming problem. Specifically, it is given as the smallest value of the variable $X^s_{f\langle\Phi\rangle^{\leq pc}f'}$ satisfying the following set of equations:

$$X^s_{f\langle\Phi\rangle^{\leq pc}f'} = \begin{cases} \sum\limits_{s \xrightarrow{\pi}_p s'} \pi \cdot X^{s'}_{f\langle\Phi\rangle^{\leq pc}f'} & \text{if } s \in S_p \\ \max(\{X^{s'}_{f\langle\Phi-\alpha\rangle^{\leq pc-\mathsf{pow}(\alpha)}f'} \mid s \xrightarrow{\alpha}_n s'\}) & \text{if } s \in S_n, s \models f \\ 1 & \text{if } s \in S_n, s \models f', \varepsilon \in \Phi, pc \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\Phi - \alpha$ is $\{\phi \mid \alpha\phi \in \Phi\}$ if $\alpha \neq \tau$ and Φ, otherwise. A solution for this set of equations can be computed as follows: for all equations of the form $X = \max\{X_1, \ldots X_n\}$, we introduce, the set of inequalities $X \geq X_i$ aiming to minimize the function $\sum_i X_i$. Using algorithms based on the ellipsoid method, this problem can be solved in time polynomial in the number of variables (see, e.g. [9]). The number of variables is $O(|T| \times 2^{|f|})$. Clearly, for each state s of T, there is one variable labeled s for each subformula of f that is considered by the algorithm. However, the number of subformulae of an *until* formula $f\langle\Phi\rangle f'$ depends on the number of regular expressions derived by the operation $\Phi - \alpha$, which is exponential in the size of Φ in the worst case. We think that it is possible to make the algorithm polynomial in the size of the formula by constructing equations differently. However, this is still left for future research.

Example. Consider two systems requiring the use of a resource. Suppose the first system employs a highly reliable resource r that never fails, $\pi(r) = 1$, but consumes a large amount of power during each of its uses. On the other hand the second system opts on using a less reliable resource r' with probability of failure 1/2 but consumes less power. The description of these systems is as follows:

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

We observe that Q attempts to use resource r' and if it is up then it consumes r', performs the event $\overline{succ}$ and returns to its initial state, otherwise, if r' is down, it retries to use r' until it succeeds. The LCMCs corresponding to processes P, Q are given in Figure 1.

We may see that although Q risks a delay in successfully using resource r', on average, it consumes less power than P per successful resource use. Specifically, it is easy to show that letting Φ be the regular expression $\{\{(r, 1, 2)\,\{(r', 1, 1)\}, \{(\overline{r'}, 1, 0)\}\}^*\ \overline{succ}$, we have that $(Q, \emptyset) \models tt\langle\Phi\rangle^{\leq 1}_{\geq 1}tt$, since, with probability 1, configuration $(Q, \emptyset)$ can eventually successfully use resource r' using 1 unit of power. On the other hand, $(P, \emptyset) \not\models tt\langle\Phi\rangle^{\leq 1}_{\geq 1}tt$, since a successful usage of resource r consumes two units of power. Introducing a time limit to the property to be checked we can see the tradeoff with respect to the time delay between using resource r' and using resource r: although $(P, \emptyset) \models tt\langle\{\Phi\rangle^{\leq 2}_{\geq 1,1}tt$, $(Q, \emptyset) \not\models tt\langle\Phi\rangle^{\leq 2}_{\geq 1,1}tt$, instead $(Q, \emptyset) \models tt\langle\Phi\rangle^{\leq 1}_{\geq 0.5,1}tt$, and $(Q, \emptyset) \models tt\langle\Phi\rangle^{\leq 2}_{\geq 0.75,2}tt$.

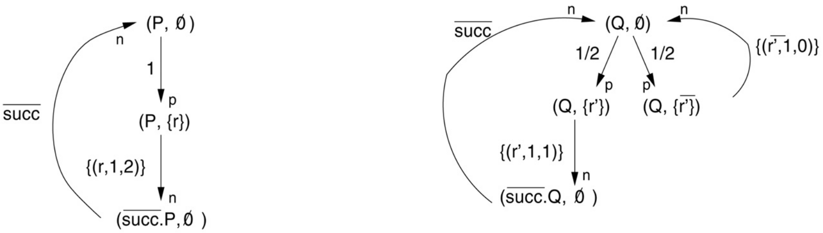

Fig. 1. The LCMCs of processes P and Q

3.2 Probabilistic Bounds on Power Consumption

Model checking P^{2}ACSR processes with respect to logical formulae allows us to verify important properties of a process. A disadvantage of this approach, however, is that to reason about power consumption we need to guess and specify a bound on power consumption in the formula. These bounds may come from the requirements for the process, but often we do not have them *a priori*. In this case, we need to compute bounds on power consumption of a process over a fixed interval of time. In this section we show how such bounds can be calculated.

Let T be an LCMC and σ a finite scheduler, where by σ being finite we mean that it can schedule a finite number of transitions. We consider the set of scheduled computations of σ and would like to compute their expected power consumption, which we denote by $Pow(T, \sigma)$. This expected value is $Pow(s_0, \sigma, \sigma_0)$, where s_0 is the initial state of T, given by the solution to the following set of equations:

$$Pow(s, \sigma, c) = \begin{cases} 0 & \sigma(c) = \perp \\ \mathsf{pow}\,(\alpha) + Pow(s', \sigma, c\alpha s') & s \in S_n, \sigma(c) = (s, \alpha, s') \\ \Sigma_{s'} \pi(s, s') \cdot Pow(s', \sigma, c\pi(s, s')s') & s \in S_p \end{cases}$$

Given a finite P^{2}ACSR process P, we may use the same scheme to compute the maximum expected power consumption $pow(P)$ over the set of all its schedulers. This can be achieved by the following algorithm for the initial state s_0. Minimum expected power consumption is obtained by replacing function max() with min(). By replacing $\mathsf{pow}\,(\alpha)$ by $\mathsf{pow}\,(\alpha, R)$, we can also compute the expected power-consumption bounds regarding the set of resources R.

```
compute_bounds( $s$ )
    if $s \in S_n$ then
        if $s$ has no outgoing transitions then $pow(s) = 0$
        else for each $s'$, $s \xrightarrow{\alpha}_n s'$
            compute_bounds( $s'$ ); $pow(s) = \max_{s \xrightarrow{\alpha}_n s'} (\mathsf{pow}\,(\alpha) + pow(s'))$;
    if $s \in S_p$ then
        for each $s'$, $s \xrightarrow{\alpha}_n s'$
            compute_bounds( $s'$ ); $pow(s) = \Sigma_{s \xrightarrow{p}_p s'} p \cdot pow(s')$
```

4 Case Study: Power-Aware Real-Time Scheduling

In this section, we describe the case study of a power-aware application, based on the work of [14] and concerning the use of dynamic voltage scaling [3] in an embedded real-time system. Dynamic voltage scaling allows to make a trade-off between performance and power consumption. A processor can lower its operating frequency, using a lower supply voltage and thus consuming less power. At the same time, a lower-frequency execution means that tasks take longer to compute. A power-aware real-time operating system has to decide when it can to operate at a lower frequency while maintaining the system's real-time requirements.

In [14], Pillai and Shin propose extensions to real-time scheduling algorithms to make use of dynamic voltage scaling. We concentrate on the extension of the Earliest Deadline First (EDF) scheduling algorithm [12] that utilizes cycles unused by the tasks to lower the operating frequency for other tasks. The algorithm assumes a set of independent periodic tasks $T_1, \ldots, T_n$ to be executed on the same processor. Each task T_i has a *period* p_i, a *worst-case execution time* c_i, and a deadline d_i by which execution must be completed. 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 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, increasing nominal utilization so that the task set may become unschedulable. However, the effective utilization may be small enough even for a lower frequency. The power-aware scheduling algorithm of [14] computes effective utilization during execution and switches frequencies to use the lowest frequency for which the task set remains effectively schedulable.

For the case study, we use the example from [14]. The task set contains three tasks with the following parameters: $c_1 = 3$, $p_1 = 8$; $c_2 = 3$, $p_2 = 10$; $c_3 = 1$, $p_3 = 14$. In each case, the deadline of the task is the same as its period. Execution times are shown for the maximum processor frequency. We assume that the processor has two possible operating frequencies. For simplicity, we assume that at the reduced frequency tasks take twice as long to execute and consume half of the power. We demonstrate that the timing constraints of the tasks are maintained even at the lower frequency and compute the savings in power consumption offered by the power-aware scheduling.

The ACSR representation of the EDF scheduling algorithm has been presented in [4]. Here, we extend that representation in P^{2}ACSR to incorporate probabilistic completion time of the tasks. An instance of the scheduling problem is modeled as a collection of processes $T_1, \ldots, T_n$. Process T_i is shown in Figure 2. A task is represented as 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, and then idles until the end of the period. If, by the end of the period, the task has not finished its execution, it will not be able accept the next $start_i$ signal, resulting in a deadlock that will signify the scheduling failure.

The other process, Job_i, upon receiving the $start_i$ signal, begins its execution. 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.

In order to model the potential for early termination, we associate, with each task, 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 terminate 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 continuation of the task execution. When the resource fails, the task terminates its execution, becoming Job_i. Otherwise, the execution continues, up to the worst-case execution time.

To model slower or faster execution of the task, depending on the operating frequency, we introduce 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 also uses two additional events, $release_i$ and $end_{i,j}$, which 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.

The algorithm of [14] recomputes effective utilization every time a task is released for execution or ends its execution. Then, it selects the least operating frequency for the processor that would still guarantee schedulability of the task set. The algorithm maintains, for each task T_i its effective utilization U_i, which is set initially, and also whenever the task T_i is released, to c_i/p_i. When T_i completes its execution for the current period, U_i is set to c_i^{act}/p_i, where c^{act} is the actual time used by the task. Every time one of the U_i values is changed, the algorithm selects the lowest operating frequency f from the set of possible frequencies such that $U_1 + \ldots + U_n \leq f/f_{max}$, where f_{max} is the highest operating frequency.

Resources used in the model of the task do not consume power since both 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, shown in Figure 3, consists of two parallel parts. The first part, represented by the process $Scale_{e_1,e_2,e_3}$, represents the voltage scaling algorithm itself. Triggered by an event $release_i$ or $end_{i,c}$ that correspond

$$
\begin{aligned}
Job_i \; &= \emptyset : Job_i + (start_i, 0).(\overline{release_i}, i).Exec_{i,0,0} \\
Exec_{i,e,t} &= e < c_i \rightarrow ((fast, i).(\{(cpu, p_{max} - (p_i - t), 0), (cont, 1, 0)\} : Exec_{i,e+1,t+1} \\
&\qquad\qquad + \{(cpu, p_{max} - (p_i - t), 0), (\overline{cont}, 1, 0)\} : (\overline{end_{i,e+1}}, i).Job_i \\
&\qquad\qquad + \emptyset : Exec_{i,e,t+1}) \\
&\qquad + (slow, i).(\{(cpu, p_{max} - (p_i - t), 0)\} : \\
&\qquad\qquad\quad (\{(cpu, p_{max} - (p_i - t), 0), (cont, 1, 0)\} : Exec_{i,e+1,t+2} \\
&\qquad\qquad\quad + \{(cpu, p_{max} - (p_i - t), 0), (\overline{cont}, 1, 0)\} : (\overline{end_{i,e+1}}, i).Job_i) \\
&\qquad\qquad + \emptyset : Exec_{i,e,t+1}) \\
&\quad + e = c_i \rightarrow (\overline{end_{i,c_i}}, i).Job_i \qquad\qquad e \in \{0..c_i\}, t \in \{0..p_i\}
\end{aligned}
$$

Fig. 2. A speed-sensitive task

to the release or, respectively, completion of the task T_i after executing for c time units, the process *SetNew* computes the effective utilization and sends signal f_{down} if a lower operating frequency is possible and signal f_{up} otherwise. The other component of the process DVS keeps the information at 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 the same resource is used with power consumption of pw_{slow}, where pw_{fast} and pw_{slow} are parameters of the model.

$$
\begin{aligned}
DVS \; &= (Scale_{c_1,c_2,c_3} \| DVS_{fast}) \backslash \{f_{up}, f_{down}\} \\[1ex]
Scale_{e_1,e_2,e_3} &= (release_1, 0).SetNew_{c_1,e_2,e_3} + (release_2, 0).SetNew_{e_1,c_2,e_3} \\
&\quad + (release_3, 0).SetNew_{e_1,e_2,c_3} \\
&\quad + \Sigma_{c \in \{1..c_1\}}(end_{1,c}, 0).SetNew_{c,e_2,e_3} + \Sigma_{c \in \{1..c_2\}}(end_{2,c}, 0).SetNew_{e_1,c,e_3} \\
&\quad + \Sigma_{c \in \{1..c_3\}}(end_{3,c}, 0).SetNew_{e_1,e_2,c} + \emptyset : Scale_{e_1,e_2,e_3} \\[1ex]
SetNew_{e_1,e_2,e_3} &= e_1/p_1 + e_2/p2 + e_3/p_3 < 1/2 \rightarrow (\overline{f_{down}}, 4).Scale_{e_1,e_2,e_3} \\
&\quad + e_1/p_1 + e_2/p2 + e_3/p_3 >= 1/2 \rightarrow (\overline{f_{up}}, 4).Scale_{e_1,e_2,e_3}) \\[1ex]
DVS_{fast} &= \{(power, 1, pw_{fast})\} : DVS_{fast} + (\overline{fast}, 1).DVS_{fast} \\
&\quad + (f_{down}, 0).DVS_{slow} + (f_{up}, 0).DVS_{fast} + \\
DVS_{slow} &= \{(power, 1, pw_{fast})\} : DVS_{slow} + (\overline{slow}, 1).DVS_{slow} \\
&\quad + (f_{down}, 0).DVS_{slow} + (f_{up}, 0).DVS_{fast}
\end{aligned}
$$

Fig. 3. P^{2}ACSR representation of voltage scaling

Analysis. We began the analysis of the case study by checking that the task set remains schedulable by the power-aware scheduling algorithm. The resulting system does not have any deadlocks, which means that all timing constraints are satisfied. Then we used the algorithm described in Section 3.2 to compute the expected power consumption of our 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 0.3, and parameters pw_{fast} and pw_{slow} were 2 and 1, respectively.

The expected minimum and maximum power consumption were 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%.

We also verified several obvious properties of the task set. For example, consider the first iteration of the first task in the set. The system, which starts in the fast mode, will switch into the slow mode if the first task finishes after the first step, which happens with probability 0.3. Executions of the other two tasks will not then affect the power mode, and the first eight steps of the system will consume only 9 units of power. All other executions will consume more power. Thus, the system satisfies the property $tt\langle power^8\rangle^{<10}_{\leq 0.3}tt$. Here, expression $power^8$ denotes eight consecutive actions that utilize the resource $power$.

5 Conclusions

We have presented P^{2}ACSR, a process algebra for resource-constrained real-time systems. The formalism allows one to model the power consumption of resources and perform power-oriented analysis of a system's behavior. We have also described two techniques for analysing P^{2}ACSR specifications. First, we have presented a probabilistic temporal power-aware logic in which one can express properties of interest regarding the behavior of power-aware, real-time systems. Second, we have presented an algorithm for computing probabilistic bounds on power consumption.

Furthermore, to allow for the automatic analysis of power-aware real-time systems, we have extended the PARAGON toolset [16], which previously allowed the specification and analysis of ACSR and PACSR processes, to handle the power consumption model of P^{2}ACSR. The toolset may accept P^{2}ACSR specifications, construct the LCMCs emanating from them, and perform model-checking as well as compute probabilistic bounds on power consumption. We have successfully applied our techniques for modeling and analysing a couple of examples, including a dynamic-voltage algorithm.

Another useful measure to be computed on P^{2}ACSR specifications which is currently being implemented in PARAGON, is that of long-run average performance. Average power consumption can be computed per unit of time or, if desired, per user-defined periods of interest, as shown in [6]. Finally, we intend to define ordering relations by which to relate processes that, although behaviorally similar, differ in their power consumption rates.

References

1. C. Baier, B.R. Haverkort, H. Hermanns, and J.-P. Katoen. On the logical characterisation of performability properties. In *Proceedings of ICALP 00*, volume 1853 of *LNCS*, pages 780–792, 2000.

2. M. Bernardo. An algebra-based method to associate rewards with empa terms. In *Proceedings of ICALP 97*, volume 1256 of *LNCS*, pages 358–368, July 1997.
3. T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In *Proceedings of IEEE Hawaii International Conference on System Sciences. Volume 1: Architecture*, pages 288–297, 1995.
4. J-Y. Choi, I. Lee, and H.-L. Xie. The specification and schedulability analysis of real-time systems using ACSR. In *Proceedings of Real-Time Systems Symposium*, December 1995.
5. E. Clarke, E. Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.
6. L. de Alfaro. How to specify and verify the long-run average behavior of probabilistic systems. In *Proceedings of IEEE Symposium on Logic in Computer Science*, pages 454–465, 1998.
7. R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. In *Proceedings of LICS '90*, 1990.
8. H. Hansson. Time and probability in formal design of distributed systems. In *Real-Time Safety Critical Systems*, volume 1. Elsevier, 1994.
9. H. Karloff. *Linear Programming.* Progress in Theoretical Computer Science. Birkhauser, 1991.
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. I. Lee, A. Philippou, and O. Sokolsky. Formal modeling and analysis of power-aware real-time systems. Technical Report MIS-CIS-02-12, Department of Computer and Information Science, University of Pennsylvania, 2002.
12. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM 20*, 1:46–61, 1973.
13. A. Philippou, O. Sokolsky, R. Cleaveland, I. Lee, and S. Smolka. Probabilistic resource failure in real-time process algebra. In *Proceedings of CONCUR 98*, pages 389–404, 1998.
14. P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of ACM Symposium on Operating Systems Principles*, 2001.
15. R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In *Proceedings CONCUR 94,* Uppsala, Sweden, volume 836 of *LNCS*, pages 481–496, 1994.
16. 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.
17. M. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 327–338, 1985.