A General Resource Framework for Real-Time Systems*

Insup Lee[1], Anna Philippou[2], and Oleg Sokolsky[3]

[1] Department of Computer and Information Science, University of Pennsylvania, USA. lee@central.cis.upenn.edu
[2] Department of Computer Science, University of Cyprus, Cyprus. annap@ucy.ac.cy
[3] Department of Computer and Information Science, University of Pennsylvania, USA. sokolsky@saul.cis.upenn.edu

Abstract. The paper describes a formal framework for designing and reasoning about resource-constrained systems. The framework is based on a series of process algebraic formalisms which have been previously developed to describe and analyze various aspects of real-time communicating, concurrent systems. We develop a uniform framework for formal treatment of resources and demonstrate how previous work fits into the new framework.

1 Introduction

An embedded system consists of a collection of components that interact with each other and with their environment through sensors and actuators. Embedded software is used to control these sensors and actuators and to provide application-dependent functionality. Two important distinguishing characteristics of embedded applications are limited resources (processing power, memory, network bandwidth, power consumption, *etc.*) and the hybrid (discrete and continuous) nature of behaviors. Many embedded systems are part of safety-critical applications, *e.g.*, avionic systems, manufacturing, automotive controllers, and medical devices.

There are two major factors that complicate the design and implementation of embedded systems. First, the software complexity of embedded systems has been increasing steadily as microprocessors become more powerful. To mitigate the development cost of software, embedded systems are being designed to flexibly adapt to different environments. The requirements for increased functionality and adaptability make the development of embedded software complex and error-prone. Second, embedded systems are increasingly networked to improve functionality, reliability and maintainability. Networking makes embedded software even more difficult to develop, since composition and abstraction principles are poorly understood.

* This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CISE-9703220, ARO DAAD19-01-1-0473, and by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

A natural response to the increasing complexity of embedded systems development is an increased emphasis on model-based development of embedded software. Models can be constructed for embedded systems and their properties can be analyzed through simulation and model checking. Models can be used to generate code skeleton and task structures and then platform dependent code can be added to work on specific environment. Since it may not be possible to completely automate code generation, models can be used to validate implementation. One way to do this is to use a design model for generating test suites, and then, use them to check the conformance of an implementation to design specifications. Another way is to extract models from legacy code and use them to validate the code with respect to specifications. The third way is to ensure that the implementation is correct at runtime through monitoring and checking of the behavior of the running system. For safety critical embedded systems, it is important to have assurance that such systems are reliable. It is well-known that activities related to certification of such systems (e.g., avionics, medical devices) are extremely time consuming and costly. Model based development can be tailored to facilitate certification processes adopted by various regulatory agencies such as FAA and FDA.

Figure 1 provides an overview of the model-based development framework being developed at the University of Pennsylvania. From the model of an embedded system specified as hybrid system, the code generator produces a set of tasks as well as code for the tasks. Given the end-to-end timing requirements of the embedded system and the description of the target hardware and operating systems platform, the timing estimator identifies the periods and deadlines of the tasks. These timing parameters are chosen to guarantee the end-to-end constraints, but the execution times of the tasks are not yet determined. The resource modeler takes the communication and synchronization structure of the generated tasks and tradeoffs between code size and execution as input and generates possible resource-aware models. From the models, the schedulability evaluator estimates the worst-case execution times and then identifies which models can be executed with their timing parameters under the available resource limits. If no such solution exists, the timing parameters of the tasks are readjusted and the design process is repeated.

In this paper, we limit our discussion to the general resource framework for embedded systems, which provides a formal semantical foundation for understanding resource-constrained behaviors subject to real-time constraints, memory limitations, power consumption, etc. The notion of a resource plays an important role in the specification and design of computing systems. It plays a central role in the domain of embedded systems, where execution is subject to a large number of resource constraints, such as timing, power consumption, size and weight, etc. We feel that, in order to properly specify and analyze such systems, a modeling formalism should incorporate the notion of a resource as a first-class entity.

Related work in the area of resource handling in embedded real-time systems falls into two categories. On the one hand, the importance of the issue

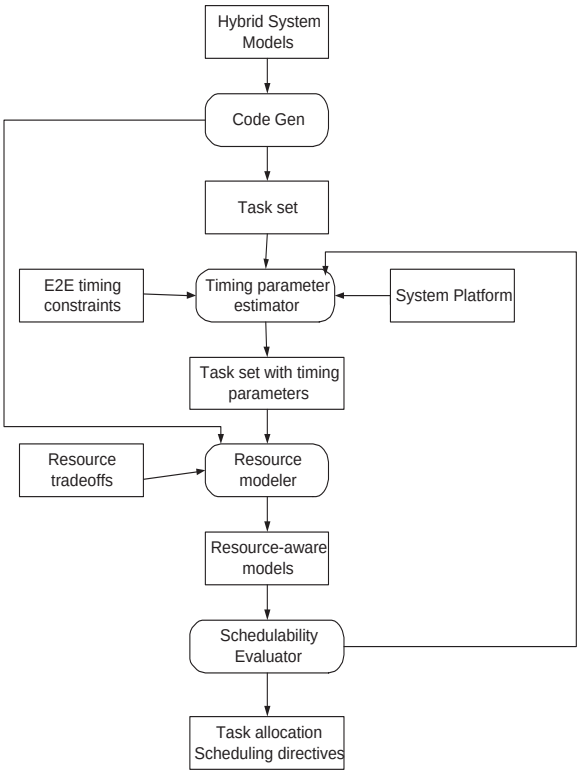

Fig. 1. Overall structure of model-based development framework

has been long realized by practitioners and a number of model-based, albeit informal, approaches have been published. We mention [13, 16, 9, 8, 3, 2] among many others. On the other hand, several formal approaches have emerged that aim at scheduling of sets of tasks under constraints. For the most part, these approaches consider only timing constraints and do not introduce the notion of resource, implicitly considering the processor as the only shared resource in the system. For example, the authors of [7] propose a formalism that allows us to model preemption in asynchronous real-time systems. In [4], the authors limit themselves to fixed-priority scheduling approaches, which allow them not to consider preemption directly, accounting for it in the worst-case computation time. The formalism of [5] provides a general scheme for handling preemption of processes due to resource contention, but the scheduling rules have to be encoded by modifying transition rules in the formalism (effectively creating a custom formalism for each scheduling policy). A different approach is taken in [1], where the authors view the scheduling activity as control and use controller synthesis techniques to model scheduled real-time systems.

In our previous work, we have proposed a family of process-algebraic formalisms for modeling and reasoning about resource-constrained systems (see [11]

for an overview). The family is built around ACSR, a discrete-time process algebra. Extensions and variations include Dense-time ACSR that includes a more general notion of time; ACSR-VP that includes a value-passing capability; PACSR, a probabilistic formalism for quantitative reasoning about fault-tolerant properties of resource-bound systems; P^2ACSR [12] that allows to specify power-constrained systems. The PARAGON toolset [18] provides tool support for modeling and analysis using these formalisms.

The family of process algebras all share the modeling approach, where a system is modeled as a collection of communicating concurrent processes. Operators to express structure of a process are also very similar in all of the formalisms. The difference lies in the way resources are used and the attributes they carry. For example, in PACSR resources have an attribute that captures the probability of failure for the resources. In ACSR-VP we can have tuples of data values associated with a resource that may be manipulated during execution, and P^2ACSR introduces power consumption attributes. In each formalism, the set of resource attributes and the way the attributes are manipulated are slightly different. This makes it difficult to systematically present the formalisms. More importantly, each extension required changes to the PARAGON toolset that have to be implemented in an *ad hoc* every time a new extension is made.

The main aim of this paper is the development of a general process algebraic framework to facilitate the construction of system models that allow us to capture faithfully all of their relevant functional and non-functional requirements. The framework allows to represent all the formalisms in the ACSR family and provide for easy incorporation of new features both into the formalism and the supporting toolset in a uniform manner. The paper presents the formal definition of the framework and demonstrate how to capture existing formalisms within the framework as well as create new ones.

2 The Framework

We define a process-algebraic formal framework for reasoning about real-time systems. The basic entity of the framework is that of a resource. We assume that a system contains a finite set of reusable resources drawn from a countably infinite set of resources $\mathcal{R}$. Resources can correspond to physical entities, such as processor units and communication channels, or to abstract notions such as message arrival.

A resource is characterized by a set of attributes that let us capture aspects of the resource's behavior depending on the needs of the application, such as timing, probabilistic, or communication behavior, or, priority and power consumption during resource usage. Resources are partitioned into classes $\mathcal{R}_1, \ldots,$ with all resources in a class having the same attributes. In turn, an attribute may have one or more elements; an attribute, a, with n elements is specified as a tuple

$$a : \langle T_1 : \mathsf{kind}_1, \ldots, T_n : \mathsf{kind}_n \rangle ,$$

where $T_1, \ldots, T_n$ are basic types such as integers, characters, tuples, etc, and $\mathsf{kind}_1, \ldots, \mathsf{kind}_n \in \{\mathsf{static}, \mathsf{dynamic}\}$ define whether the value of the attribute's

element remains constant throughout computation (static) or is associated to every resource use (dynamic).

Example 1. As an example, consider the class of resources $\mathcal{R}$ that may experience failures, consume power, and whose use is regulated by priorities, We may characterize this class by three attributes as follows:

$$\mathcal{R} : [\pi : \langle [0,1] : \mathsf{static} \rangle, pc : \langle int : \mathsf{dynamic} \rangle, pr : \langle int : \mathsf{dynamic} \rangle].$$

Attribute π captures the possibility of resource failure. It has one element, that of the probability of failure, which is assumed to be constant throughout all executions of the resource. Attribute *pc* is the power consumption, which may be different in each resource use depending on the level of power required on each occasion, and *pr* is the priority of a resource access, which may also be different depending on which process uses r.

When writing a model in the framework, given a resource r, we specify the values of all of the static elements of its attributes, and then, whenever r is used in the model, it is accompanied by values for all of the dynamic elements of its attributes. Furthermore, we write $a_r(i)$ for the ith element of attribute a of resource r. Given resources *cpu* and *chan* in the resource class $\mathcal{R}$, we specify once initially that, for example, $\pi_r(cpu) = 0.01$ and $\pi_{chan}(1) = 0.1$. Then, whenever each of the resources is used in the model, we give the values of its dynamic attributes in a *resource access*, for example, $(cpu, 2.5, 1)$. We will always assume positional correspondence between the dynamic attribute names in the attribute tuple of a resource class and the values of attributes in the resource use. Therefore, in $(cpu, 2.5, 1)$, $pc = 2.5$ and $pr = 1$. Resource accesses are specified in *actions*. An action A is a collection of resource accesses. By $\rho(A)$ we denote the multiset of names of resources used in A. Actions are building blocks for processes.

Syntax. We let P, Q range over processes, A ranges over actions, and I ranges over sets of resources. The following grammar describes the syntax of processes and actions.

$$\begin{aligned} P &::= \text{ NIL} \mid A : P \mid P + Q \mid P \| Q \mid [P]_I \mid P \backslash\backslash I \\ A &::= \{(r_1, a_{11}, \ldots, a_{1n_1}), \ldots, (r_m, a_{m1}, \ldots, a_{mn_m})\} \end{aligned}$$

Process NIL represents the inactive process. Process $A : P$, is the prefix operator: it executes action A and proceeds to P. 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 actions. The construct $[P]_I$, $I \subseteq \mathsf{R}$, referred to as *resource closure*, produces a process that reserves the use of resources in I for itself, extending every action A in P with resources in $I - \rho(A)$. Finally, $P \backslash\backslash I$, referred to as *resource hiding*, allows the process to *hide* or *restrict* the identity of resources in I so that they are not visible on the interface with the environment of process P.

Example 2. As an example of a process, consider

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

where resources *cpu* and *chan* are drawn from the resource class $\mathcal{R}$ of Example 1. Process P represents a processor that can accept messages from a channel. We assume that reading the message from the channel requires additional power than remaining idle. Depending on whether the message arrives or not, P has two alternative behaviors. If the message arrives, as described in resource access $(chan, 1, 0)$, the processor may receive the message, consuming 3 units of power, and proceed to process it as P_1. Otherwise, the processor consumes only 1 unit of power and continues as P_2.

Semantics. The semantics of the framework is given operationally by a transition system that captures the behavior of processes. It is based on the notion of a configuration which comprises of a process and a world/state that can be used to keep useful information about the resources of the process. We write $\mathcal{C}$ for the set of all configurations and we write $S(P) \in \mathcal{C}$, for a configuration containing process P in state S. Finally, we write Act for the set of all actions a configuration can engage in. The semantics is based on a function

$$F(\mathcal{C}, Act) \longrightarrow 2^{\mathcal{C}}$$

which, given a process configuration $S(P)$ and an action A, returns the set of configurations that can be reached from $S(P)$ by performing action A. Thus, the semantics is based on the following rule:

$$\frac{S'(P') \in F(S(P), A)}{S(P) \stackrel{A}{\longrightarrow} S'(P')}$$

Domain specialization. In order to adjust the general framework for the needs of a specific application domain, we must give meaning for resources and their attributes and establish the semantics for the processes. The following steps are needed to perform the specialization for a particular domain.

- *Resource classes.* A finite set of resource classes need to be established for the domain along with the attributes of the class.
- *Syntactic consistency.* A predicate *valid* must be provided. For a given action A, $valid(A)$ denotes that the action can be legitimately used in a model within this domain. Furthermore, we may restrict the domain for the set of resources I appearing in process constructs $[P]_I$ and $P \backslash\backslash I$.
- *Semantic interpretation.* Finally, the set $\mathcal{C}$ of configurations has to be defined and the function F has to be given.

3 Framework instantiations

In this section, we will show how to instantiate the general framework to several progressively more complex domains.

3.1 CCS

The first domain we consider is the CCS domain [14]. CCS processes consider only communication constraints between concurrent processes. The actions that processes can engage in are *send* or *receive* messages on named channels. In addition, there is a *silent* action denoted τ. A send action and a receive action on the same channel can synchronize to produce a silent action, whereas the silent action cannot synchronize with any other action. We introduce two resource classes, $\mathcal{R}_1$ and $\mathcal{R}_2$. The class $\mathcal{R}_1$ has one attribute *polarity* of type $\langle\{!,?\} : \mathsf{dynamic}\rangle$. The class $\mathcal{R}_2$ does not have attributes and contains the single resource τ. As a shorthand, and to coincide with CCS style, we write $r!$ for $\{(r,!)\}$, the send action on resource (channel r), $r?$ for $\{(r,?)\}$ the receive action on channel r and τ for $\{(\tau)\}$ the silent action.

The consistency predicate for an action A stipulates that A is valid if and only if $\rho(A)$ is a singleton. Finally, we require that in $[P]_I$, I ranges from the empty set of resources, i.e. the process construct is disabled, whereas in $P\backslash\backslash I$, I can be any subset of the resource class $\mathcal{R}_1$, that is, we can only hide named channels.

The process does not need any additional state information ($S(P) = P$), and the semantic function is defined recursively on the process structure, following the standard CCS approach. We begin by considering how actions interact with the process constructs. Let A and B be well-formed actions and $I \subseteq \mathcal{R}_1$, then we define:

$$A\|B = \begin{cases} \tau, & \text{if } \{A,B\} = \{a?, a!\} \\ \bot, & \text{otherwise} \end{cases}$$

$$A\backslash\backslash I = \begin{cases} A, & \text{if } \rho(A) \notin I \\ \bot, & \text{otherwise} \end{cases}$$

Consequently we have that two actions may be composed in parallel to produce the silent action if they are send and receive actions on the same channel, and that an action can survive the hiding operator only if does not involve a resource from set I.

We may now define the semantic function F as follows, where + stands for summation $\mathsf{mod}\,2$.

$$\begin{array}{lll} F(A{:}P, A) = \{P\} & & \\ R \in F(P_1 + P_2, A) & \textit{iff} & \exists i \in \{1,2\} \text{ such that } R \in F(P_i, A) \\ R \in F(P_1 \| P_2, A) & \textit{iff} & (P_1 \xrightarrow{A_1} P_1',\ P_2 \xrightarrow{A_2} P_2',\ A = A_1 \| A_2, \\ & & R = P_1' \| P_2') \text{ or } (\exists i \in \{1,2\} \text{ such that} \\ & & P_i' \in F(P_i, A),\ R = P_i' \| P_{i+1}) \\ R \in F(P\backslash\backslash I, A) & \textit{iff} & R' \in F(P, B),\ A = B\backslash\backslash I,\ R = R'\backslash\backslash I \end{array}$$

3.2 ACSR

ACSR can be viewed as an extension of CCS with time-consuming steps and priorities. We add a new resource class $\mathcal{R}_3$ with the attribute *time* of type

$\langle int : \mathsf{static} \rangle$ and specify that for any resource $r \in \mathcal{R}_3$, $time_r = 1$. That is all time-consuming actions take one unit of time. In addition, all three classes have the attribute *priority* of type $\langle int : \mathsf{dynamic} \rangle$.

The consistency predicate states than an action A is well-formed if the resources occurring in A, $\rho(A)$, are pairwise distinct and, satisfy either $\rho(A) \subseteq \mathcal{R}_3$, in which case they are referred to as *timed actions*, or $\rho(A) = \{r\}$, $r \in \mathcal{R}_1 \cup \mathcal{R}_2$, in which they are referred to as *instantaneous events*. We write $\mathcal{D}_R$ for the set of timed actions and $\mathcal{D}_E$ for the set of instantaneous events. As before, we omit the set brackets from actions involving resources in $\mathcal{R}_1 \cup \mathcal{R}_2$, and simply write $(a!, p)$ and $(a?, p)$ for send and receive actions along channel a. Further, we specify that, in $[P]_I$, $I \in 2^{\mathcal{R}_3}$, and that, in $P\backslash\backslash I$, $I \in 2^{\mathcal{R}_1} \cup 2^{\mathcal{R}_3}$.

We now proceed to give the semantic function for ACSR. This is similar to the one in the CCS domain, except that it handles timed actions and, further, applies the preemption relation $\prec$, which specifies when two actions are comparable with respect to priorities. For example, $\emptyset \prec A$ for all $A \in \mathcal{D}_R$, that is, the idle action $\emptyset$ is preemptable by all other timed actions, and $(a, p) \prec (a, p')$, whenever $p < p'$. For the precise definition of $\prec$ we refer to [10].

There is no state information and the definition of the semantic function is similar to that of CCS. We begin by describing the composability of actions with the various operators. Let A and B be well-formed actions, $I \in 2^{\mathcal{R}_3}$, and $J \in 2^{\mathcal{R}_1} \cup 2^{\mathcal{R}_3}$, then

$$A \| B = \begin{cases} (\tau, p + p'), & \text{if } \{A, B\} = \{(a?, p), (a!, p')\} \\ A \cup B & \text{if } A, B \in \mathcal{D}_R,\ \rho(A) \cap \rho(B) = \emptyset \\ \bot, & \text{otherwise} \end{cases}$$

$$[A]_I = \begin{cases} A \cup \{(r, 0) \mid r \in I - \rho(A)\}, & \text{if } A \in \mathcal{D}_R \\ A, & \text{otherwise} \end{cases}$$

$$A\backslash\backslash J = \begin{cases} \{(r, p) \in A \mid r \notin J\}, & \text{if } A \in \mathcal{D}_R \\ A, & \text{if } A \in \mathcal{D}_E,\ \rho(A) \notin J \\ \bot, & \text{otherwise} \end{cases}$$

We point out that two timed actions may be composed together only if the resources they access are independent from each other, that is, they do not compete for the use of any common resources. In case of the contrary, $A \| B = \bot$, signifying that a deadlock arises. $[A]_I$ and $A\backslash\backslash I$ capture the informal explanation of the process constructs $[P]_I$ and $P\backslash\backslash I$, respectively. The former ensures that access to resources I is reserved for process P by employing all of the resources $r \in I - \rho(A)$ at priority level 0. As a result any further sharing of the resources in I, with any parallel process of P, is prohibited (see the definition of $A \| B$). The latter disables any instantaneous events involving a channel in I and hides the use of I-resources from timed actions.

We proceed to define function F. We let A, B range over the set of actions Act of ACSR, consisting of timed actions and instantaneous events.

$$\begin{array}{lll} F(A{:}P, A) = \{P\} & & \\ R \in F(P_1 + P_2, A) & \textit{iff} & \exists i \in \{1, 2\} \text{ such that } R \in F(P_i, A) \\ & & \text{and, if } P_{i+1} \xrightarrow{B}, \ A \not\prec B \end{array}$$

$$
\begin{array}{lll}
R \in F(P_1 \| P_2, A) & \textit{iff} & [(P_1' \in F(P_1, A_1),\ P_2' \in F(P_2, A_2), \\
 & & A = A_1 \| A_2,\ R = P_1' \| P_2') \\
 & & \text{or } (A \in \mathcal{D}_E, \text{ and } \exists i \in \{1,2\} \cdot \\
 & & P_i' \in F(P_i, A) \text{ and } R = P_i' \| P_{i+1})] \\
 & & \text{and, if } \exists i \in \{1,2\} \cdot P_i \xrightarrow{B}, B \in \mathcal{D}_E \\
 & & \text{or } P_1 \xrightarrow{B_1}, P_2 \xrightarrow{B_2}, B = B_1 \| B_2, \\
 & & \text{then } A \not\prec B \\
R \in F([P]_I, A) & \textit{iff} & R' \in F(P, A'),\ R = [R']_I,\ A = [A']_I \\
 & & \text{and, if } P \xrightarrow{B} \text{ then } A \not\prec [B]_I \\
R \in F(P \backslash\backslash I, A) & \textit{iff} & R' \in F(P, A'),\ R = R' \backslash\backslash I,\ A = A' \backslash\backslash I, \\
 & & \text{and, if } P \xrightarrow{B} \text{ then } A \not\prec B \backslash\backslash I
\end{array}
$$

The first two rules define the semantics of the prefix and summation operators. The third rule describes the behavior of the parallel composition operator. This allows component processes to proceed independently or synchronize with one another with respect to instantaneous actions and forces processes to synchronize on timed actions, making timed transitions truly synchronous, in that a process only advances if both of its subprocesses take a step. By the definition of $A_1 \| A_2$ we have that only one process may use a resource during any time step. The next rule describes the behavior of the close operator: When a process is embedded in a closed context, such as $[P]_I$, we ensure that there is no further sharing of the resources $r \in I - \rho(A)$ by employing all of these resources at priority level 0 (see definition of $[A]_I$). Instantaneous events are not affected by the close operator. Finally, the rule for resource hiding establishes that the set of resources I is restricted from the interface with the environment. Note, that in all but the first rule, a side condition checks that the action in question cannot be preempted by any other enabled action of the process.

3.3 PACSR

The PACSR domain is aimed at fault-tolerance analysis of real-time systems. Resources are allowed to fail with a fixed probability during an execution. This is captured by extending $\mathcal{R}_3$ with an additional attribute π of type $\langle [0,1] : \mathsf{static} \rangle$, representing a probability. This probability captures the rate at which the resource may fail. To be able to reason about failed as well as non-failed resources, we also have the attribute *status* of type $\langle \{up, down\} : \mathsf{dynamic} \rangle$. Intuitively, the process $\{(r, 1, up)\} : P$ will succeed in performing action $\{(r, 1, up)\}$ with probability π_r and fail, becoming NIL with probability $1 - \pi_r$. On the other hand, the process $\{(r, 1, down)\} : P$ will fail with probability π_r, exactly when the first process succeeds, and succeeds with probability $1 - \pi_r$. The use of failed resources is useful when we need to specify recovery from failures. We adopt the following notation: for all $r \in \mathcal{R}_3$, we write (r, p), for $(r, p, up))$, and $(\overline{r}, p)$, for $(r, p, down)$.

The consistency condition for the PACSR domain is the same as for the ACSR domain, both in case of the validity predicate and in the case of the syntactic conditions for resource hiding and resource closure.

We continue to define the semantics function F for PACSR. As already mentioned, resources are associated with a probability of failure. Thus, the behavior of a system has certain probabilistic aspects to it which must be reflected in the operational semantics of the domain. For example consider action $\{(cpu, 2), (chan, 1)\}$, where resources *cpu* and *chan* have probabilities of failure 0 and $1/3$, respectively, that is $\pi_{cpu} = 1$ and $\pi_{chan} = 2/3$. Then the action takes place with probability $\pi_{cpu} \cdot \pi_{chan} = 2/3$ if both resources are up, and fails with probability $1/3$ if either of the resources fails. Therefore, behavior of a given process P depends on the status of resources which are relevant to P. To capture information about resource status in the configuration we write $S(P)$ for process P in state S, where $S \in 2^{\mathcal{R}_3} \times \{up, down\}$ records the status of resources of P. Configurations are partitioned into probabilistic configurations, from which only probabilistic steps are possible that update resource failure information, and non-deterministic configurations, where the state of all relevant resources is known and transitions can be computed.

The intuition for the semantics is as follows: for a process P, we begin with the configuration $\emptyset(P)$. As computation proceeds, probabilistic transitions are performed from configurations to determine the status of probabilistic resources immediately relevant for execution (denoted $\mathsf{imr}(P)$) but for which there is no knowledge in the configuration's world. Once the status of a resource is determined by some probabilistic transition, it cannot change until the next timed action occurs. Once a timed action occurs, the state of resources has to be determined anew, since in each time unit resources can fail independently from any previous failures. Nondeterministic transitions (which can involve events or actions) are performed from nondeterministic configurations. Precise definition for $\mathsf{imr}(P)$ is given in [15].

Let $S = \{(r_1, s_1), \ldots, (r_n, s_n)\} \subseteq \mathcal{R}_3 \times \{(up, down)\}$ and $I = \{r_1, \ldots, r_n\} \subseteq \mathcal{R}_3$. We write

- $\mathsf{p}(S) = \Pi_{1 \leq i \leq n, s_i = up}\ \pi_{r_i} \cdot \Pi_{1 \leq i \leq n, s_i = down}\ (1 - \pi_{r_i})$,
- $\mathcal{W}(I) = \{\{(r_1, s_1), \ldots, (r_n, s_n)\} \mid s_i \in \{up, down\}\}$, and
- $res(S) = I$.

We partition the set of configurations into the sets of nondeterministic configurations, $\mathcal{C}_N$, and probabilistic configurations, $\mathcal{C}_P$. We have that $S(P) \in \mathcal{C}_N$ *iff* $\mathsf{imr}(P) - res(S) = \emptyset$, that is, there is no immediate resource of P whose status is not already recorded in S, and $S(P) \in \mathcal{C}_P$, otherwise. We proceed to define function F.

$S'(P) \in F(S(P), \ell)$	*iff*	$P \in \mathcal{C}_P$, $S'' \in \mathcal{W}(\mathsf{imr}(P) - res(S))$, $S' = S \cup S''$ and $\ell = \mathsf{p}(S'')$
$F(S(A{:}P), A) = \{S(P)\}$	*iff*	$S(A{:}P) \in \mathcal{C}_N$ and $A \in \mathcal{D}_E$
$F(S(A{:}P), A) = \{\emptyset(P)\}$	*iff*	$S(A{:}P) \in \mathcal{C}_N$, $A \subseteq S$ and $A \in \mathcal{D}_R$
$R \in F(S(P_1 + P_2), A)$	*iff*	$S(P_1 + P_2) \in \mathcal{C}_N$, $\exists i \in \{1, 2\}$ such that $R \in F(S(P_i), A)$ and if $S(P_{i+1}) \xrightarrow{B}$, $A \prec B$
$R \in F(S(P_1 \| P_2), A)$	*iff*	$S(P_1 \| P_2) \in \mathcal{C}_N$, and [$S'(P_1') \in F(S(P_1), A_1)$,

$S'(P_2') \in F(S(P_2), A_2)$,
$A = A_1 \| A_2$, $R = S'(P_1' \| P_2')$
or
$A \in \mathcal{D}_E$, and $\exists i \in \{1,2\} \cdot$
$S'(P_i') \in F(S(P_i), A)$, $R = S'(P_i' \| P_{i+1})$]
and,
if $\exists i \in \{1,2\} \cdot S(P_i) \xrightarrow{B}$, $B \in \mathcal{D}_E$
or $S(P_1) \xrightarrow{B_1}$, $S(P_2) \xrightarrow{B_2}$, $B = B_1 \| B_2$,
then $A \not\prec B$

$R \in F(S([P]_I), A)$ *iff* $S([P]_I) \in \mathcal{C}_N$, $S'(Q) \in F(S(P), A')$,
$A = [A]_I$, $R = S'([Q]_I)$, and,
if $S(P) \xrightarrow{B}$ then $A \not\prec [B]_I$

$R \in F(S(P \backslash\backslash I), A)$ *iff* $S(P \backslash\backslash I) \in \mathcal{C}_N$, $S'(Q) \in F(S(P), A')$,
$A = A' \backslash\backslash I$, $R = S'(Q \backslash\backslash I)$, and,
if $S(P) \xrightarrow{B}$ then $A \not\prec B \backslash\backslash I$

Thus, given a probabilistic configuration $S(P)$, with I the immediate resources of P for which the state is not yet determined in S, and $S'' \in \mathcal{W}(I)$, P enters the state extended by S'' with probability $\mathsf{p}(S'')$. Note that configuration $S(P)$ evolves into $S'(P)$ which is, by definition, a nondeterministic configuration.

Example 3. To illustrate the probabilistic transition relation, consider process

$$P \stackrel{\text{def}}{=} \{(r_1, 1), (r_2, 2)\} : P_1 + (e, 1).P_2$$

in the initial configuration $\emptyset(P)$. 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 $\mathcal{C}_p$, from where we have four probabilistic transitions that determine the states of r_1 and r_2:

$$\emptyset(P) \xrightarrow{\pi_{r_1} \cdot \pi_{r_2}} \{r_1, r_2\}(P), \qquad \emptyset(P) \xrightarrow{\pi_{r_1} \cdot (1-\pi_{r_2})} \{r_1, \overline{r_2}\}(P),$$
$$\emptyset(P) \xrightarrow{(1-\pi_{r_1}) \cdot \pi_{r_2}} \{\overline{r_1}, r_2\}(P), \quad \text{and} \quad \emptyset(P) \xrightarrow{(1-\pi_{r_1}) \cdot (1-\pi_{r_2})} \{\overline{r_1}, \overline{r_2}\}(P).$$

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

The remaining of the rules concerning the nondeterministic configurations follow along the same lines as the ACSR rules. The only point to note concerns the prefix operator: for the timed action A to be performed by configuration $S(A : P)$, it must be that all resources in A are available in the configuration's state.

Example 4. Returning to the previous example, the nondeterministic configuration $\{r_1, r_2\}(P)$, where $P \stackrel{\text{def}}{=} \{(r_1, 2), (r_2, 2)\} : P_1 + (e, 1).P_2$ has two nondeterministic transitions:

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

The other configurations $\{r_1, \overline{r_2}\}(P)$, $\{\overline{r_1}, r_2\}(P)$, and $\{\overline{r_1}, \overline{r_2}\}(P)$, allow only the e-labeled transition since either r_1 or r_2 is failed.

4 Multi-capacity resources and memory constraints

In this section we use the resource framework to construct a formalism MCSR that captures memory use as a different kind of resource. Memory is a critical resource in size-constrained embedded systems such as mobile phones. In the design of an embedded system, we need to consider tradeoffs between memory use and the speed of the tasks in the system. A task can be made to use less memory at the cost of executing longer. But this increased execution can violate timing constraints in the system. The proposed formalism will allow us to capture such tradeoffs and reason about their effects.

We see that the nature of memory as a resource is different from other serially reusable resources considered so far in this paper. Two processes can use the same memory, as long as the total use does not exceed the memory capacity. Therefore, we introduce the new class of resources called *multi-capacity* resources.

We develop MCSR as an extension of ACSR (see Section 3.2) by adding a new resource class, $\mathcal{R}_4$, of multi-capacity resources. We will use resources in this class to represent memories, however, other kinds of resources may be modeled in the same way. Each resource has two attributes. The first attribute, *capacity* of type $\langle int : static \rangle$, represents the capacity of the resource. The second attribute, *use* of type $\langle int : dynamic \rangle$, captures the memory used in a step of a process.

The consistency predicate in this framework, extends that of ACSR by allowing multi-capacity resources to be used in timed actions of MCSR processes, so that for a timed action A, $\rho(A) \subseteq \mathcal{R}_3 \cup \mathcal{R}_4$. The semantic function for MCSR is the same as for ACSR, except that the definition of action composition is modified as follows:

$$A \| B = \begin{cases} (\tau, p + p'), & \text{if } \{A, B\} = \{(a?, p), (a!, p')\} \\ A \uplus B, & \text{if } A, B \in \mathcal{D}_R,\ (\rho(A) \cap \rho(B)) \cap \mathcal{R}_3 = \emptyset \text{ and} \\ & \forall r \in \mathcal{R}_4, (r, u_1, c) \in A, (r, u_2, c) \in B \Rightarrow u_1 + u_2 \leq c \\ \bot, & \text{otherwise} \end{cases}$$

The operator $A \uplus B$, defined on compatible timed actions (meaning that $A \| B \neq \bot$), is defined as follows:

1. $(r, a_1, \ldots, a_n) \in A \uplus B$ if $r \in \mathcal{R}_3 \cup \mathcal{R}_4, (r, a_1, \ldots, a_n) \in A$ and $r \notin \rho(B)$ or $(r, a_1, \ldots, a_n) \in B$ and $r \notin \rho(A)$, or
2. $(r, u_1 + u_2, c) \in A \uplus B$ if $r \in \mathcal{R}_4, (r, u_1, c) \in A$ and $(r, u_2, c) \in B$.

Memory aware scheduling. We illustrate the use of MCSR by showing a collection of periodic tasks that execute within the same system, sharing the processor and memory. Each task T_i is characterized by its period p_i and execution time c_i. Each task is released for execution at the beginning of every period and its deadline is equal to the period. That is, a task has to use the processor for c_i time units in each interval $[k * p_i, (k + 1) * p_i]$, for each integer k. Tasks are assigned a priority for accessing the processor, and an executing task can be preempted by a task with a higher priority. Each task has a fixed amount $m_{i,c}$ of memory allocated to store the code and static data structures. In addition,

when the task is released for execution, an additional amount of memory, $m_{i,d}$, is allocated to the task for keeping its dynamic data. Once the task completes its execution in the current period, the dynamically allocated memory is released.

To model such a task in MCSR, we adapt the approach of [6], extending it with additional resources representing memory consumption. To simplify presentation, we assume that priorities of tasks are fixed, for example, according to rate-monotonic scheduling discipline. More complex dynamic-priority scheduling approaches, such as EDF, can be accomodated as well. An instance of the scheduling problem is modeled as a collection of processes $T_1, \ldots, T_n$. Process T_i is shown below. A task is represented as a parallel composition of two processes: Job_i and $Activator_i$.

$$
\begin{array}{lll}
T_i & = (Job_i \| Activator_i) \backslash\backslash \{start_i\} & \\
Activator_i & = (\overline{start_i}, i).\emptyset^{p_i} : Activator_i & \\
Job_i & = \{(mem, m_{i,c})\} : Job_i + (start_i, 0).Exec_{i,0,0} & \\
Exec_{i,e,t} & = e < c_i \rightarrow \{(cpu, i), (mem, m_{i,c} + m_{i,d})\} : Exec_{i,e+1,t+1} & \\
 & \quad + \{(mem, m_{i,c} + m_{i,d})\} : Exec_{i,e,t+1} & \\
 & \quad + e = c_i \rightarrow Job_i & e \in \{0..c_i\}, t \in \{0..p_i\}
\end{array}
$$

The description of task T_i involves the use of three resources: $start_i \in \mathcal{R}_1$, $mem \in \mathcal{R}_4$ and $cpu \in \mathcal{R}_3$, the last two corresponding to the system's processor and available memory. We assume a value M for the capacity of multi-capacity resource *mem*. The task consists of two processes running in parallel. 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, Job_i begins its execution. At each time step, the task has a fixed priority that is equal to the task index. 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 higher priority. In this case, the interrupted task executes a timed action that does not use the *cpu* resource, but retains its memory use. In this case, its accumulated execution time stays the same while the elapsed time is increased.

Note that each job uses some amount of the memory resource *mem* in each step. However, this amount is different in different states of the task. While the job is waiting for the $start_i$ signal in Job_i, it uses $m_{i,c}$ units of memory, and after the task is started, it uses $m_{i,c} + m_{i,d}$ units of memory, regardless of whether it is running or preempted.

Code size vs. execution time tradeoff modeling. In a recent paper [17], Shin *et al.* study a different problem that considers a tradeoff between code size

(static memory use) and execution time. By choosing different encodings of the instructions sets, modern embedded processors offer the possibility to reduce the code size at the expense of a longer execution time. Each task can have its own encoding. However, increased execution time may render the task set unschedulable, while increased code size may exceed the memory capacity. Thus we have to find an encoding for each task that will satisfy all the constraints. For a task i, the tradeoff is represented as a list of pairs $(m_{i,j}, c_{i,j})$, where the first element is the code size and the second element is the respective execution time. The encoding is chosen statically before the task begins executing.

We can address this problem with a modified task model shown above. Each task initially makes a non-deterministic choice from J possibilities, choosing its memory use $m_{i,j}$ and execution time $c_{i,j}$. Since the memory use remains constant once the choice has been made, only the first step of the process $JobStart_i$ uses the memory resource. The rest of the job process consists of J disjoint copies of the process Job_i from the previous example, for the different values of $c_{i,j}$. When the task processes are combined in parallel to represent the complete system, some of the initial choices become infeasible, exceeding the memory constraints. Of the initial choices that satisfy the memory constraints, some will violate the timing constraints during execution, resulting in a deadlock in the state space. Therefore, to identify parameter values that satisfy both timing and memory constraints, analysis will have to identify initial steps that lead to deadlock-free subsystems.

$$
\begin{array}{lll}
T_i & = (JobStart_i \| Activator_i) \backslash \{start_i\} & \\
Activator_i & = (\overline{start_i}, i).\emptyset^{p_i} : Activator_i & \\
JobStart_i & = \Sigma_{j \in J}\{(mem, m_{i,j})\} : Job_{i,j} & \\
Job_{i,j} & = \emptyset : Job_{i,j} + (start_i, 0).Exec_{i,j,0,0} & \\
Exec_{i,j,e,t} & = e < c_{i,j} \rightarrow \{(cpu, i)\} : Exec_{i,j,e+1,t+1} & \\
 & \quad + \emptyset : Exec_{i,j,e,t+1} & \\
 & + e = c_{i,j} \rightarrow Job_{i,j} & e \in \{0..c_{i,j}\}, t \in \{0..p_i\}
\end{array}
$$

5 Conclusions

We have presented a formal approach to the design of real-time embedded systems, which explicitly captures resource constraints that affect the system behavior. The approach includes an extension mechanism that allows us to easily incorporate new kinds of resources and resource constraints. We have shown how to model memory capacity constraints using multi-capacity resources. The resource-modeling formalism is incorporated into a larger model-based development framework for embedded systems.

We are working to identify additional classes of resources and develop means of incorporating them into the formalism, as well as providing flexible tool support for the model development in the formalism. An interesting extension to the current work is to go beyond serially reusable resources to *consumable* resources, which can be used only a fixed amount of times during a computation and possibly replenished after a certain amount of time passes.

References

1. K. Altisen, G. Goessler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems*, 23:55–84, 2002.
2. H. Ammar, V. Cortellessa, and A. Ibrahim. Modeling resources in a UML-based simulative environment. In *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA '01)*, June 2001.
3. L. Baum and T. Kramp. Towards a uniform modeling technique for resource-usage scenarios. In *Proceedings of PDPTA '99*, June-July 1999.
4. V.A. Braberman and M. Felder. Verification of real-time designs: combining scheduling theory with automatic formal verification. In *Proceedings of the 7th European Engineering Conference*, pages 494–510, 1999.
5. M. Buchholtz, J. Andersen, and H.H. Loevengreen. Towards a process algebra for shared processors. In *Workshop on Models for Time-Critical Systems*, BRICS Notes Series NS-01-5, pages 87–99, August 2001.
6. J-Y. Choi, I. Lee, and H-L Xie. The specification and schedulability analysis of real-time systems using ACSR. In *Real-Time Systems Symposium*. IEEE Computer Society Press, December 1995.
7. J. Ermont and F. Boniol. TPAP: an algebra of preemptive processes for verifying real-time systems with shared resources. In *Workshop on Theory and Practice of Timed Systems*, April 2002.
8. A. Agrawal *et al.* MILAN: A model based integrated simulation framework for design of embedded systems. In *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001)*, June 2001.
9. E. Huh, L. Welch, B. Shirazi, and C. Cavanaugh. Heterogeneous resource management for dynamic real-time systems. In *Heterogeneous Computing Workshop*, pages 287–296, May 2000.
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, J.-Y. Choi, H.-H. Kwak, A. Philippou, and O. Sokolsky. A family of resource-bound real-time process algebras. In *Proceedings of 21st International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'01)*, August 2001.
12. I. Lee, A. Philippou, and O. Sokolsky. Process algebraic modelling and analysis of power-aware real-time systems. *Computing and Control Engineering Journal*, to appear, 2002.
13. A. Mehra, A. Indiresan, and K.G. Shin. Resource management for real-time communication: Making theory meet practice. In *Proceedings of IEEE Real-Time Technology and Applications Symposium*, pages 130–138, June 1996.
14. R. Milner. *Communication and Concurrency*. Prentice Hall Intl., 1989.
15. A. Philippou, O. Sokolsky, R. Cleaveland, I. Lee, and S. Smolka. Probabilistic resource failure in real-time process algebra. In *Proceedings CONCUR 98*, pages 389–404. Springer-Verlag, 1998.
16. S. Saewong and R. Rajkumar. Cooperative scheduling of multiple resources. In *IEEE Real-Time Systems Symposium*, pages 90–101, 1999.
17. I. Shin, I. Lee, and S.L. Min. Embedded system design framework for minimizing code size and guaranteeing real-time requirements. In *Proceedings of IEEE Real-Time Systems Symposium*, December 2002. To appear.
18. 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.