May 2004

Resource Modeling for Embedded Systems Design

Oleg Sokolsky

University of Pennsylvania, sokolsky@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_papers

Recommended Citation

Resource Modeling for Embedded Systems Design

Abstract
The paper describes a formal framework for designing and reasoning about resource-constrained embedded 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 concurrent systems. We present a uniform framework for formal treatment of resources and illustrate modeling of common resource classes.

Keywords
Model-based design, resource modeling, process algebra

Comments
Publisher URL: http://dx.doi.org/10.1109/WSTFES.2004.1300422

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.
Resource Modeling for Embedded Systems Design*

Oleg Sokolsky
University of Pennsylvania

Abstract

The paper describes a formal framework for designing and reasoning about resource-constrained embedded 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 concurrent systems. We present a uniform framework for formal treatment of resources and illustrate modeling of common resource classes.

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. An important distinguishing characteristic of embedded applications is limited resources (processing power, memory, network bandwidth, power consumption, etc.). Many embedded systems are part of safety-critical applications, e.g., avionic systems, manufacturing, automotive controllers, and medical devices. Their reliability has to be properly assured.

At the same time, complexity of embedded systems has been progressively increasing due to more powerful microprocessors and wider use of networking. A natural response to the increasing complexity 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 later be used to validate the implementation.

Figure 1 provides an overview of our model-based development framework. The development process begins with system requirements that specify high-level behavior of the system, and a model of the system environment. They are combined into a single model that captures the system functionality with respect to a given environment. The system contains a collection of concurrent components. Components can communicate with each other during execution, representing a wide variety of data dependencies and precedence requirements. Such a model, however, does not take resource constraints into consideration. Every concurrent process that is ready to execute, can proceed forward. From this behavioral model we extract a set of tasks that capture the structure of components in the model, abstracting away computation details. The task set is then enriched with information about resource types and constraints. Each computational step of a task is annotated with required resources. If a resource that is needed by a step is shared with another task, the two tasks cannot proceed concurrently. On this model, we perform schedulability and timing analysis of the system to ensure that, for a given allocation of resources to components, timing requirements of the system are satisfied and no resource conflicts arise. If a resource conflict is found, a new resource allocation has to be used, or more resources need to be added to the system. In addition, we can perform analysis of behavioral properties of the model to ensure that functional requirements are satisfied. Once model analysis is performed to a satisfactory degree, we can proceed with the implementation of the system, driven by the resource-aware model. Automatic code

* This research was supported in part by NSF CCR-0086147, NSF CCR-0209024, ARO DAAD19-01-1-0473
generation can create a skeleton of the system implementation, while test generation provides a test suite that demonstrates the compliance of the implementation to the model.

In this paper, we limit our discussion to the resource-handling part of the framework, highlighted in Figure 1. We use a formalism that 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 a central role in the specification and design of embedded systems, where execution is subject to various 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. Other aspects of the development framework are described in [2, 3, 10].

Related work in the area of resource modeling in embedded real-time systems falls into two categories. On the one hand, the importance of the issue has been long realized by practitioners and a number of model-based, albeit informal, approaches have been published. We mention [15, 17, 11, 4] 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 [6, 7, 9]. 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 [13] for an overview). The family is built around ACSR, a discrete-time process algebra. Extensions and variations include ACSR-VP that includes a value-passing capability; PACSR, a probabilistic formalism for quantitative reasoning about fault-tolerance properties; P²ACSR [20] specifying power-constrained systems. The PARAGON toolset [19] provides tool support for modeling and analysis using these formalisms. The formalisms share the modeling approach, where a system is modeled as a collection of communicating concurrent processes, and offer similar modeling constructs. 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 are manipulated during execution, and P²ACSR has power consumption attributes. In each formalism, the set of resource attributes and the way the attributes are manipulated are slightly different. We outline 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 captures all the formalisms in the ACSR family and allows us to uniformly incorporate new features both into the formalism and PARAGON in a uniform manner.

Acknowledgements. I would like to thank Insup Lee of the University of Pennsylvania and Anna Philippou of the University of Cyprus for fruitful collaboration in the development of the framework.

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 resources. A resource is characterized by a set of attributes, such as timing parameters, priority, power consumption; probabilistic or communication behavior. Resources are partitioned into classes. All resources in a class have the same attributes. An attribute \( a \) of a class is specified as \( a : \langle T : \text{kind} \rangle \), where \( T \) is a basic type such as integers, characters, tuples, etc, and \( \text{kind} \in \{ \text{stat}, \text{dyn} \} \) denoting whether the value of the attribute’s element remains constant throughout computation (stat) or is associated to every resource use (dyn). We write \( a_r \) for the attribute \( a \) of resource \( r \).

Example. As an example, consider the class of resources \( R_f \) that may experience failures, consume power, and whose use is regulated by priorities. We may characterize this class by three attributes as follows:

\[
R_f : [\pi : \{ [0, 1] : \text{stat} \}, \text{pc} : \{ \text{int} : \text{dyn} \}, \text{pr} : \{ \text{int} : \text{dyn} \}].
\]

Attribute \( \pi \) captures the possibility of a 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 \( \text{pc} \) is the power consumption, which may be different in each resource access depending on the requested operation, and \( \text{pr} \) is the priority of the access, which may also be different depending on which process uses \( r \).

When we use a resource \( r \) in a model we specify once the values of all 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 in a tuple called resource access. 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. If we have resources \( \text{cpu} \) and \( \text{chan} \) in the resource class \( R_f \), we specify initially that, for example, \( \pi_{\text{cpu}} = 0.01 \) and \( \pi_{\text{chan}} = 0.1 \). An access of the resource \( \text{cpu} \) may be \( (\text{cpu}, 2.5, 1) \). 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 names of resources used in \(A\). Actions are building
blocks for processes.

Syntax. Below, \(P, Q\) range over processes, \(A\) ranges over
actions, and \(I\) ranges over sets of resources. The following
grammar describes the syntax of processes and actions.

\[
P :: \quad \text{NIL} \mid A : P \mid P + Q \mid |P||Q| \mid [P]_f \mid P\setminus I
\]

Process NIL represents the inactive process. Process \(A : P\)
executes action \(A\) and proceeds as \(P\). Process \(P + Q\) represen-
ta non-deterministic 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
\([P]_f\), \(I \subseteq R\), 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
\(I \setminus \rho(A)\). Finally, \(P\setminus I\), 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. As an example of a process, consider

\[
P \triangleq (\text{cpu, 3, 2, } (\text{chan, 1, 0})) : P_1 + (\text{cpu, 1, 1}) : P_2.
\]

where resources \(\text{cpu}\) and \(\text{chan}\) are drawn from the resource
class \(R\) 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 ar-
rials or not, \(P\) has two alternative behaviors. If the mes-
essage arrives, as described in resource access (\(\text{chan, 1, 0}\)),
the processor may receive the message, consuming 3 units
of power, and proceed to process it as \(P_1\). Otherwise, the
processor consumes 1 unit of power and continues as \(P_2\).

Semantics. The semantics is given operationally by a trans-
ition 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 \(C\)
for the set of all configurations and we write \(S(P) \in C\), for a
configuration containing process \(P\) in state \(S\). Finally, \(Act\)
is the set of all actions a configuration can engage in. The se-
manics is based on a function

\[
F(C, Act) \rightarrow 2^C
\]

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:

\[
S'(P) \in F(S(P), A)
\]

\[
S(P) \xrightarrow{A} S'(P)
\]

Domain specialization. In order to adjust the gen-
eral framework for the needs of a specific application
domain, we must give meaning for resources and their at-
tributes and establish the semantics for the processes. The
following steps perform the specialization for a particu-
lar domain: 1) Establish a finite set of resource classes for
the domain, along with the attributes of the class. 2) Pro-
vide the syntactic consistency predicate \(\text{valid}\). For a given
action \(A\), \(\text{valid}(A)\) denotes that the action can be legiti-
mately used in a model within this domain. Furthermore,
we may restrict the domain for the set of resources \(I\) ap-
pearing in process constructs \([P]_f\) and \(P\setminus I\). 3) Define the
set \(\mathcal{C}\) of configurations and the function \(F\).

3. Resource classes

We consider a number of resource class definitions that
are useful for modeling embedded systems. Class defi-
nitions include serially reusable shared resources, used
to model processor units, communication resources, used
to model synchronous and asynchronous communication
channels, and multi-capacity resources that naturally
 correspon-
d to memory modules. For each class of re-
sources, we specify the set of attributes and define the sem-
antic function \(F(C, Act)\) for a formalism that uses this
resource class. Note that if a formalism uses multiple re-
source classes, the semantic function should also deal
with actions that contain resources of several kinds. Func-
tions for several such formalisms are given in [14].

Serially reusable resources. The basic kind of resources
that is necessary to consider for schedulability analysis of
real-time systems are serially reusable resources. While
such a resource is used by a process, no other process can
access it. As soon as the process releases the resource, it is
immediately available for use by another process. It is a na-
tural abstraction for a processor unit. One task is scheduled
to execute on a processor, and any other task that is also
ready to run is blocked until the processor becomes avail-
able. Tasks access processors according to their priorities.
Different tasks can access processor resources with differ-
ent priorities, therefore priority is a dynamic attribute of a
serially reusable resource. We will assume here that all re-
sources are used for one time unit. A more complex defini-
tion can be given, where a resource access has another at-
tribute, access duration.

The resource class is defined as \(\mathcal{R}_s = [pr : \langle int : \text{dyn} \rangle]\).
The consistency predicate \(\text{valid}(A)\) states that an action \(A\)
 is well-formed if the resources occurring in \(A\), \(\rho(A)\), are
pairwise distinct. The semantic function uses the following
auxiliary functions:

\[
[A]_f = \begin{cases} A \cup \{r, 0\} & \text{if } r \in I \setminus \rho(A) \\ \bot & \text{otherwise} \end{cases}
\]

\[
| A | B = \begin{cases} A \cup B & \text{if } A, B \in \mathcal{D}_R, \rho(A) \cap \rho(B) = \emptyset \\ \bot & \text{otherwise} \end{cases}
\]
$A \setminus J = \{ (r,p) \in A \mid r \notin J \}$

$A \parallel B$ ensures that two timed actions may be composed together only if they do not compete for the use of any common resources. Otherwise, $A \parallel B = \bot$, signifying that a deadlock arises. $[A]_I$ reserves access to resources $I$ for the process performing $A$ by employing all of the resources $r \in I - \rho(A)$ at priority level 0. As a result, resource in $I$ cannot be shared with any other process (according to $A \parallel B$). $A \setminus J$ hides the use of $J$-resources from timed actions. We also use the preemption relation on actions. Intuitively, action $A_2$ preempts action $A_1$, written $A_1 \prec A_2$, if $A_2$ has a resource $r$ that has a higher priority in $A_2$ than in $A_1$, and $A_1$ does not have such resources. There is no need to store extra state information, so for each configuration $S$, $S(P) = P$.

$F(A;P,A) = \{ P \}$

$R \in F(P, +, A)$ iff $\exists \in \{ 1, 2 \}$ such that $R \in F(P, A)$ and if $F(P_{i+1}, B) \neq \emptyset$, then $A \not\prec B$

$R \in F(P, || P, A)$ iff $\{ P \} \in F(P, A)$ and if $F(P_{i+1}, || P) \neq \emptyset$, then $A \not\prec B$

$R \in F([P], A)$ iff $R' \in F([P], A')$, $R' \not\prec A$, $A = A'$, and if $F([P], B) \neq \emptyset$, then $A \not\prec B$

$R \in F(P \setminus I, A)$ iff $R' \in F(P, A')$, $R' \not\prec I$, $A = A' \setminus I$, and if $F(P \setminus I, B) \neq \emptyset$, then $A \not\prec B$

Communication resources. We introduce a resource class that allows us to specify input and output actions. We will use this distinction in the semantic function to define a resource class that can communicate in an internal step. We introduce a resource class $R_c = [p : \{ ?, !, x \} : \text{dyn}]$. The class has one dynamic attribute that indicates the polarity of resource access, distinguishing between an input action on channel $a'$ (denoted $a'$) instead of $\{ a, a' \}$, an output action on channel $a$, denoted $a$, and an internal step that involves a busy channel $a$, denoted $a_x$. The consistency predicate stipulates that a communication action can contain only one resource of the class $R_c$ and no other resources. The semantic definition uses the following function:

$\alpha_x$ if $\{ A, B \} = \{ a', a \}$ otherwise

Thus two actions may be composed in parallel if they are send and receive actions on the same channel, and the composition results in a busy channel step. Composition of any other communication actions is undefined.

Asynchronous communication resources. To model asynchronous buffered communication between processes, we can use the following resource class. Contents of messages are not modeled, however processes can store or retrieve several messages in one step. The resource class is described as $R_b = [p : \{ ?, ! \} : \text{dyn}, \text{size} : \{ \text{int} : \text{dyn} \}, e : \{ \text{int} : \text{stat} \}]$. The static attribute $c_r$ represents the capacity of resource $r$, that is, the maximum number of messages that can be stored in the buffer. In order to define the semantic function for such resources, we need to maintain state information that keeps, for each channel, the number of messages stored in the buffer of the channel. Using $S(r)$ for the number of messages in the resource $r$ in state $S$, the following clause of $F(S(P), A)$ deals with communication actions:

$F(S(A; P), A) = \{ S'(P) \}$ iff either $A = \{ (r, n), S(r) = m, m - n \geq 0, S'(r) = m - n, S'(r') = S(r'), r \neq r', A = S(n), S(r) = m, m + n \leq c_r, S'(r') = m + n, S'(r') = S'(r'), r \neq r' \}$. This definition represents the natural intuition that a process can read from a channel if the channel has enough messages and can write to a channel if there are enough empty slots to write all the messages. With these definitions we can obtain a formalism similar to communicating real-time state machines of [18].

Resources with failures. For probabilistic reasoning about fault tolerance of resource-constrained systems, we have defined a class of resources $R_m$ that can experience transient failures with a fixed probability [16, 14]. It extends $R_m$ with an additional attribute $\pi$ of type $\{ [0, 1] : \text{stat} \}$, representing the probability of the resource failing in the next step. To be able to reason about failed as well as non-failed resources, we also have the attribute $\text{status}$ of type $\{ \{ \text{up}, \text{down} \} : \text{dyn} \}$. The process $\{ (r, 1, \text{up}) \} : P$ will succeed in using $r$ with probability $\pi$, or fail, becoming NIL, with probability $1 - \pi$. Replacing $\text{up}$ with $\text{down}$, we get the process that fails exactly when the other one succeeds. Failed resources are useful to specify failure recovery.

Combining resource classes. When multiple resources are used in the same formalism, the definitions need to be extended to handle the case when an action contains resources of multiple classes. For example, the process algebra ACSR [12] employs serially reusable resources and a variant of synchronizing communication resources. However, from the point of view of the formalism, synchronization on communication channels happens instantaneously, while access of other resources takes time. Therefore, the syntactic consistency predicate includes a provision that resources of the two classes cannot appear in the same action.

4. Analysis

We can employ a number of different techniques to analyze models in the resulting formalisms. Analysis of timing behavior for systems using serially reusable resources has been performed in [5]. More generally, similar techniques can be used to analyze functional correctness with respect to arbitrary safety properties. Schedulability analysis for a variety of scheduling algorithms has been described in [8]. Probabilistic analysis of timing and fault tol-
herence properties has been considered in [16]. An extension of that approach to power-aware systems has been considered in [20]. Analysis techniques include model checking with respect to a temporal logic that can express power consumption constraints, as well as computation of probabilistic bounds on power consumption. We can also compute minimum and maximum power consumption over a set of executions. Power-aware scheduling has been modeled and analyzed in [20]. The resource-based approach also allows us to explore a number of design trade-offs. For example, we have considered trade-offs between performance and power consumption [20] and between performance and memory consumption [14].

5. Conclusions and future work

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 defined several resource classes for commonly encountered modeling scenarios. 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 direction is to consider 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. Such an extension will allow us to have a more natural treatment of battery-operated devices.

References


