In Proc. of the 12th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (ISORC), Tokyo, Japan. March 2009.
Revised version.

Resource Scopes: Toward Language Support for Compositional Determinism

Madhukar Anand
Cisco Systems
anandmkr@cisco.com

Sebastian Fischmeister
Dept. of Elect. and Comp. Eng.
University of Waterloo
sfischme@uwaterloo.ca

Insup Lee
Dept. of Comp. and Inf. Science
University of Pennsylvania
lee@cis.upenn.edu

Abstract

Complex real-time embedded systems should be compositional and deterministic in the resource, time, and value domains. Determinism eases the engineering of correct systems and compositionality simplifies the assembly of complex systems out of smaller modules. This paper describes the PEACOD framework that is developed to support deterministic behavior for resource consumption, value passing, and timing. The paper introduces the notions of determinism in the context of the resource, value, and temporal domains, and present the resource-scope language construct that can be used to program such deterministic behaviors. Furthermore, the paper also provides semantics for the resource scope construct and uses these semantics to show that the program behavior is preserved under composition. The paper briefly describes the current implementation of PEACOD.

1. Introduction

Predictability is a desirable system property. General systems should, for example, be predictable with respect to value behavior; given a set of operations and inputs, the developer should be able to predict possible results of these operations. Real-time embedded systems should not only be predictable with respect to values, they should also be predictable with respect to timing and resource consumption. For one, embedded systems usually have tight resource constraints in terms of memory and computation, and real-time systems must meet specified deadlines for input/output behavior, for instance, sampling sensors and actuating motors.

One goal is to make a system predictable with respect to a property, such as timing, is to make the system deterministic with respect to this property. *Determinism* means that an external observer can consistently predict the system's future state including resource consumption, input/outputs, and timing. For the value domain this means that an observer can predict the future output values. For resources, this means predicting the system's resource consumption over time.

Compositionality is another desirable property that enables composing a system from smaller parts. Embedded software grows in functionality and complexity and composing a system is one well-known and studied method to cope with this. *Compositionality* means that the properties of the complex system are determined by the properties of its constituent modules and the rules used to combine them. Similar to predictability, a system can be compositional with respect to properties. We will inspect the value, resource, and time domain.

In general, compositionality provides many new challenges and existing research looks at compositionality under different directions, e.g., on the one hand, Shin et al. [1] and Wandeler et al. [2] view it from the point of execution time and scheduling (i.e., resources and timing). On the other, Giotto [3] investigates compositionality with respect to value behavior and timing. Each of these models/systems make assumptions about the domains left out in the model (e.g., ignoring value behavior or assuming unlimited resources), and this can cause problems when realizing a system with these models such as too much resource overhead or correct scheduling but incorrect value behavior resulting from vacant sampling. To date, no system offers one model that provides a unified framework for the resource, value, and temporal domain. In this work, we introduce our framework and its main programming construct, the resource scope, proposed to fill this gap.

PEACOD (Penn's Abstraction for Compositional Determinism) is a framework to simplify the development of complex real-time embedded systems by providing determinism and compositionality in the three mentioned domains. It also includes a programming construct that enables the developer to specify the behavior in the three domains.

In our framework, we extend the temporal scopes [4] to *resource scopes*. A resource scope specifies the encapsulated program's behavior in the time, value, and resource domain. In the time domain, the scope allows specifying temporal constraints similar to temporal scopes. In the value domain, the scope allows specifying input/output values. And in the

This work was partly sponsored by AFOSR FA9550-07-1-0216, NSF CNS-0721541, NSF CNS-0720703, NSERC DG 357121-2008, and by CIMIT under U.S. Army Medical Research Acquisition Activity Cooperative Agreement W81XWH-07-2-0011. The information contained herein does not necessarily reflect the position or policy of the Government, and no official endorsement should be inferred.

resource domain, the system allows declaring resource demands necessary for the successful execution of the scope's mandatory part. Section 6 elaborates on this and introduces the language construct for resource scopes.

Our notion of determinism is based on similar runs with respect to a specific view. Whenever a scope is executed, it generates a run that follows a specific behavior with respect to timing, value behavior, and resource consumption. Informally, a system is deterministic with respect to a property, if that property holds each time it is executed with the same (sequence of) input. For example, a system always reacts to a particular event within five milliseconds. The five millisecond bound on the reaction time is the property and if the system is deterministic with respect to this property, then the system will always react in less than five milliseconds given the same input. Section 4 establishes this concept of similar runs and defines what is necessary to specify determinism in the time, value, and resource domain on the basis of similar runs.

The PEACOD framework guarantees that resource scopes behave the same way each time they execute w.r.t. timing, value behavior, and resource consumption. In Section 6.2, we specify the behavior of resource scopes in PEACOD and prove in Section 3 that this behavior is met and preserved under composition. For sake of brevity, we only specify PEACOD's behavior in terms of events, because a formal description of the operational semantics is too long for this format. Finally, we provide an implementation outline in Section 7 on the basis of resource, value, and time mediators that control each domain for the set of active, running scopes. We also discuss how related work (Section 8) provides approaches for determinism in a subset of the three domains and put PEACOD in context of the related work.

2. Model and Assumptions

We make several assumptions in our framework to achieve the objectives of determinism and compositionality. A real-time embedded system consists of a set of resources and an embedded program that consumes them during its execution. We assume deterministic resources, which requires resources to be measurable, reliable, and provide a continuous supply in multiples of the their supply quantum. Each resource comes with the necessary infrastructure to implement our framework such as a programming interface.

We assume that an embedded program consists of a set of periodic tasks. These tasks are preemptible and consist of a period, an offset, and a deadline. The deadline can be earlier than the period, but not later. We assume a static task set with known arrival times. Tasks consume resources such as CPU cycles for computation. A task can consume a resource in a time-shared or exclusive fashion. To eliminate precedence in resource consumption, at any time one task accesses at most one resource in time-shared and all other in exclusive fashion. We assume that each task's resource consumptions are measurable, bounded, known a priori, and in multiples of the resources' quantum.

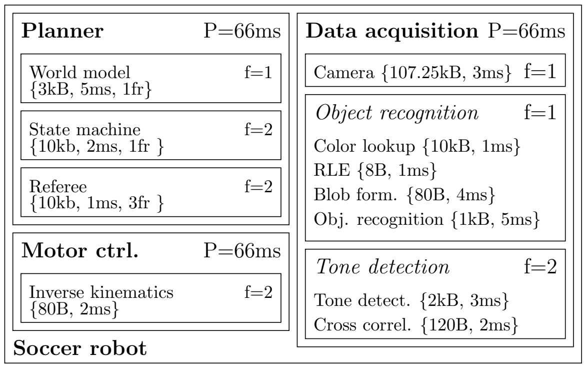

Figure 1. Software modules of a soccer robot.

We assume that time is given in discrete units. Further, we assume the presence of a global clock and all times are measured with respect to this clock.

We note that our approach is not necessarily to support the fast execution of a real-time program or optimal resource utilization but to make it predictable. Further, the framework allows the programmer to develop independent software modules and compose them. It is our aim to provide compositionality in the program's value, time, and resource domain.

3. Example

We use soccer playing robots [5] as an example to guide through the concepts. A team of AIBO dogs is programmed to play soccer. One team tries to score a goal, which is defended by the opposing team and vice versa.

Figure 1 shows a subset of a soccer robot's software modules and their resource consumptions; for a full description see [5]. The application consists of three parts: the planner, the motor control, and the data acquisition. Each part consists of modules, which must complete within the part's period (e.g., the planner executes every 66 milliseconds and must run the world-model and the state-machine module). Next to the modules' names, we list their resource consumption, e.g., the world-model modules requires three kilobytes of memory, five milliseconds of full CPU time (corresponding to roughly 600,000 instructions), and sends one frame via the AIBO's wireless link. We also list the module's frequency, e.g., the world module runs once every period.

The functionality is as follows: the planner makes tactical decisions, the world-model module locates the robot and objects on the field, the state-machine module determines future actions, and the referee module implements the referee-control protocol. The motor control steers the motors and the inverse-kinematics module translates Cartesian space into joint space. The data acquisition collects data and prepares information for the planner; the camera module senses a 208×176 image, the object-recognition module processes the image and detects/recognizes objects, the tone-detection module identifies sound commands by other team members.

The soccer robot example above represents a real-time embedded application composed of multiple modules. The modules communicate via a shared medium and several modules together work towards a common goal. A desired characteristic for such a setting is compositionality and predictability, since it allows developing the modules independently, composing them at the end, and keeping the original properties, such as timing constraints. We understand that the soccer robot is only a toy example, however, in large systems such as an airplane or a car, several vendors independently implemented components are composed and distributed across different electronic control units. Regarding compositionality, companies face the similar challenges as in the compositional robot soccer example.

4. Definitions

In this section, we define the underlying terminology, which is necessary for our understanding of determinism introduced in Section 4.3. We will illustrate the concepts with examples from the soccer robot. However, we omit the units from all quantities for better presentation of the main ideas.

4.1. Resources

A *resource* R is defined by the tuple $\langle q, supply\rangle$ where (1) q is the quantum of utilization, and (2) $supply$ is a quadruple (c_R, p_R, o_R, d_R) such that the resource is supplied for c_R units every p_R time units in the interval $(o_R, d_R]$. In the soccer robot example, which is implemented on a 120 MIPS processor, we have the resource CPU $cpu_r = \langle 1, (120, 1, 0, 1)\rangle$, the resource memory $mem_r = \langle 1, (64 * 14, 1, 0, 1)\rangle$, the resource Ethernet bus $bus_r = \langle 1, (70\,000, 1, 0, 1)\rangle$.

A *resource consumption* rc is defined by the tuple $\langle q, demand\rangle$ where (1) q is the quantum of utilization and (2) $demand$ is a quadruple $(c_{rc}, p_{rc}, o_{rc}, d_{rc})$ where $c_{rc} = k \cdot q, k \in \mathbb{Z}$ is resource demand every p_{rc} units of time in the time interval $(o_{rc}, d_{rc}]$.

A *consumption configuration* cc is a set of resource consumptions RC and CC is the set of all possible resource consumptions, $\mathcal{P}(RC)$. The CPU consumption for the camera module is, $cpu_{cam} = \langle 1, (3, 44, 0, 44)\rangle$, the memory consumption $mem_{cam} = \langle 1, (81, 44, 0, 44)\rangle$. The consumption configuration of the camera module is $cc_1 = \{cpu_{cam}, mem_{cam}\}$ and another is $cc_\perp = \{\}$ (an example of the idle consumption). For this case, $CC_{cam} = \{\{cc_1, cc_\perp\}, \{cc_1\}, \{cc_\perp\}\}$

4.2. Tasks and Systems

Thus far, we have defined resources, their supply and consumption. Now we define how consumption is used in tasks and composed into systems.

A task τ is defined as tuple $\langle I, O, c, p\rangle$ where (1) I are input variables, (2) O are output variables, (3) a consumption c, and (4) a period p. For example, the camera module task has a 208×176 pixel array as output, the period is $44ms$ and the earlier given consumption.

An embedded program is typically composed of several tasks. A *connector* tells the runtime environment how different tasks are connected. The concept of a connector is analogous to hardware connectors where the output of one component are connected to the input of another component.

We define the *connector* as $con = \langle o, i\rangle$ where (1) o is a task's output variable and (2) i is a task's input variable. Given the camera module τ_{cam} with output variables $o_1 \ldots, o_{208\times176}$ and the object-recognition module τ_{or} with the input variables $i'_1 \ldots, i'_{208\times176}$, we can connect them by a series of connectors, $\langle (o_1, i'_1), \ldots, (o_{208\times176}, i'_{208\times176})\rangle$. Note that several connectors can connect one task's output variable with other task's input variables.

We define a *system* as a set of tasks. To compose a system from several parts, we need an abstraction to represent the composed system. We will use the task abstractions to represent the whole system's resource requirements. The task abstraction of the system is composed from the individual resource requirements of its constituent tasks.

We define *resource abstraction* as follows: Given a set of resource consumptions $\langle q, demand\rangle$, an abstraction is a tuple $\langle q_A, demand_A\rangle$ such that q_A is the minimum quantum of utilization and $demand_A$ is such that it preserves the timing and resource demands of all the underlying demands. For a more elaborate and formal discussion see [1].

In the example below, we have manually computed abstractions and they are presented for purposes of illustration only.

Example 1. *Consider the CPU demands of the data acquisition module of the robot-soccer example. The Object Recognition module (ORM) is composed of Color Lookup (CL), Run-length Encoding (RLE), Blob Formation (BF) and Object Recognition (OR) sub-tasks. Let us first get an abstraction for the ORM module from each sub-task. For this, we start with the resource demand functions for the CPU of the four sub-tasks:* $(1, 66, 0, 6)$ *for the CL,* $(1, 66, 6, 12)$ *for the RLE,* $(4, 66, 12, 36)$ *for the Blob Formation and* $(5, 66, 36, 66)$ *for the OR. A possible supply of* $(1, 3, 0, 3)$ *can meet these demands and hence is an abstraction of the sub-tasks (see Section 5 on how to prove this).*

(a) Run tr_1.

clk	I	O	cc
0	0	0	$cc_\perp$
5	1	0	cc_1
10	1	1	cc_1
15	1	1	cc_2
20	1	1	$cc_\perp$

(b) Run tr_2.

clk	I	O	cc
0	0	0	$cc_\perp$
4	1	1	$cc_\perp$
5	1	1	cc_1
15	1	1	cc_2
20	1	1	$cc_\perp$

(c) Run tr_3.

clk	I	O	cc
0	0	0	$cc_\perp$
5	1	0	cc_1
7	1	0	cc_2
10	1	1	cc_1
20	1	1	$cc_\perp$

(d) Run tr_4.

clk	I	O	cc
0	0	0	$cc_\perp$
8	1	0	cc_1
12	1	1	cc_1
17	1	1	cc_2
20	1	1	$cc_\perp$

Table 1. Example runs.

4.3. Determinism

We now define the notion of determinism applicable to values, resources, and time. We first define the task state and run, which subsequently allows us to compare the different tasks' runs to check for deterministic behavior. For sake of brevity, we use the abstract example of Table 1. It shows four runs of the same task and we use it to discuss determinism.

A *task state* ts of a task τ is defined as $\langle clk, \mathcal{V}_i, \mathcal{V}_o, cc\rangle$ with (1) a clock value clk, (2) a valuation of input variables $\mathcal{V}_i$, (3) a valuation of output variables $\mathcal{V}_o$, and (4) a consumption configuration cc. Given a task τ_1, with one input and one output, one possible task state is $ts(\tau_1) = \langle 5, 1, 0, cc_1\rangle$ (see Table 1(a)).

A *task run* tr defines a total order on the set of task states for a given task τ. ts_{i+1} is the successor state of ts_i denoted via $ts_i \vdash ts_{i+1} \vdash ts_{i+2}$. The successor state must differ from the original state by at least one element other than the clock value; otherwise, any two task states have an infinite number of task states between them that are different by clock values. Note that the task state remains valid over time until the successor state.

A *view* $v \subseteq \{clk, I, O, cc\}$ is a projection of a task state ts on to certain elements of the tuple. We denote the projection as $v(ts)$. For example, a view could ignore inputs and only show the elements clk, O, cc of task states.

Any two runs tr and tr' are said to be *similar* with respect to a selected view v denoted $tr \sim_v tr'$ if at any time t, the two task states defined by the runs are equivalent with respect to the view v: $\forall t : v(\langle t, \mathcal{V}_i, \mathcal{V}_o, cc\rangle) = v(\langle t, \mathcal{V}'_i, \mathcal{V}'_o, cc'\rangle)$ where $\langle t, \mathcal{V}_i, \mathcal{V}_o, cc\rangle$ results from run tr and $\langle t, \mathcal{V}'_i, \mathcal{V}'_o, cc'\rangle$ results from run tr'. We can easily check similarity, since we only need to check similarity whenever one run changes task state.

Now, based on the definition of similar runs, we can state that if all runs of a task are similar to each other w.r.t. the attributes of a view (e.g., values only, or values and timing only), then it is deterministic w.r.t. those attributes.

Definition 1. *(Types of Determinism) For a system producing runs tr, tr', we define the following types of determinism $tr \sim_v tr'$ with the corresponding view v:*

- *value determinism with $v = \{I, O\}$,*
- *time determinism with $v = \{clk\}$,*
- *resource determinism with $v = \{cc\}$,*
- *value–time determinism with $v = \{I, O, clk\}$,*
- *resource–time determinism with $v = \{cc, clk\}$,*
- *resource–value determinism with $v = \{I, O, cc\}$, and*
- *value–resource–time det. with $v = \{clk, I, O, cc\}$.* □

Example 2. *Table 1 shows four task runs. Runs tr_1 and tr_2 are resource–time deterministic, tr_1 and tr_3 are value–time deterministic, tr_1 and tr_4 are resource–value deterministic; remember that timing does not matter for this type of determinism. On the contrary, tr_1 and tr_3 are* not *resource–time deterministic, because tr_1 changes to cc_2 at time 15 and tr_3 changes to cc_2 at time 7.*

5. Schedulability

So far, we defined the terminology, the notions of determinism and an abstraction with respect to values, time and resources. We now briefly describe how we can calculate the abstractions and check schedulability, to ensure that sufficient resources are available for individual resource scopes. Note that we apply existing approaches.

Example 1 shows that PEACOD needs to compute an abstraction for composing systems. In this section, we outline how we compute this abstraction and check schedulability under such an abstraction.

The problem of checking feasibility of a task set that includes asynchronous periodic tasks is co-NP complete [6]. To simplify the analysis, we therefore suggest using a periodic resource model [1] to get a synchronous periodic abstraction for the set of tasks, which can then be checked for feasibility in polynomial (linear) time.

Notation: we denote a task τ as a quadruple (c, p, o, d) where c is the execution time, d is the relative deadline, p is the period, and o, the offset. We assume that $d \leq p$. We assume that we are given a set of tasks $T = \{\tau_1, \ldots, \tau_n\}$ to be scheduled. We denote $H = lcm(p_1, \ldots, p_n)$ and $\Phi = \max(o_1, \ldots, o_n)$.

Checking feasibility of scheduling tasks in T is co-NP-complete in the strong sense. This was originally shown by Leung and Merill [6].

Theorem 1. *The feasibility analysis for task systems T is co-NP-complete in the strong sense.* □

Baruah et al [7] gave a feasibility test for EDF which does not require computing the entire schedule. Their condition is based on *Processor Demand Function*, η where $\eta_i(t_1, t_2)$

is defined as the total number of natural numbers k such that,

1) $t_1 \leq kp_i + o_i$ (a release occurs at or after time t_1) and
2) $o_i + kp_i + d_i \leq t_2$ (the corresponding deadline falls at or before t_2).

More precisely, we can say that, $\eta_i(t_1, t_2) = \max(\left\lfloor \frac{t_2 - d_i - o_i}{p_i} \right\rfloor - \left\lceil \frac{t_1 - o_i}{p_i} \right\rceil + 1, 0)$

Theorem 2. *([7]) EDF produces a valid schedule for T, a task set with integer valued parameters iff,*

1) $U(T) \leq 1$
2) $df(t_1, t_2) = \sum_{i=1}^{n} \eta_i(t_1, t_2) \leq t_2 - t_1$ *for all* $0 \leq t_1 < t_2 \leq 2H + \Phi$

□

The above condition was shown to be equivalent to checking all offset-deadline pairs in [1]. We will therefore use this condition to check for feasibility of our resource scopes.

A periodic resource model for compositional real-time guarantees was proposed in [1]. We will use that model to compute a composable abstraction for the tasks in T. The periodic resource model $\Gamma(\Pi, \Theta)$ is a partitioned resource that guarantees allocations of Θ time units every Π time units where the resource period Π is a positive integer and Θ is a real number in $(0, \Pi]$. A *supply bound function* (sbf) is defined as the minimum resource supply of Γ during a time interval t as follows: $\mathbf{sbf}_\Gamma(t) = \left\lfloor \frac{t-(\Pi-\Theta)}{\Pi} \right\rfloor \cdot \Theta + \epsilon_s$ where, $\epsilon_s = \max\left(t - 2(\Pi - \Theta) - \Pi \left\lfloor \frac{t-(\Pi-\Theta)}{\Pi} \right\rfloor, 0\right)$.

With the supply function, we can now synthesize an abstraction for multiple tasks. This is done by choosing a periodic resource model which will meet the demands of the underlying tasks. For further details, we refer the reader to Shin et al. [1]. Such an abstraction allows us to reason about the composed system and check for schedulability without having to dig into each of the underlying tasks.

Example 3. *In Example 1 we noted that* $(1, 3, 0, 3)$ *is an abstraction for the object recognition module. The Tone detection module consists of tone detection and cross correlation tasks. These sub-tasks have demands* $(3, 22, 0, 13)$ *and* $(2, 22, 13, 22)$*, respectively. One possible abstraction for these tasks is* $(1, 4, 0, 4)$*. The camera module demands* $(3, 66, 0, 66)$*, the world module demands* $(5, 50, 0, 50)$*, the state-machine module demands* $(2, 22, 0, 22)$*, the referee module demands* $(1, 22, 0, 22)$*, and finally the inverse-kinematics module demands* $(2, 33, 0, 33)$.

With these abstractions, every task in the system has a periodic resource demand which we can use to check the overall schedulability of the system by an EDF scheduler: The utilization is, $(\frac{1}{3} + \frac{1}{4} + \frac{3}{66} + \frac{5}{50} + \frac{2}{22} + \frac{1}{22} + \frac{2}{33}) = 0.9257 \leq 1$ *and therefore it is schedulable. Note that we can use the principle for other resources, because at most one resource is time shared while the others are used exclusively.*

6. PEACOD Language Construct

So far, we defined determinism and our abstraction to compose systems. Now, we present a language construct and its semantics and prove that it provides value–resource–time determinism and also compositional determinism. We assume that a high-level structuring construct is given that allows us to assemble resource scopes sequentially (operator ;) and concurrently (||) by using the specified identifier (e.g., $sc1||sc2$). We do not elaborate on sequential composition, because it does not add to the discussion of determinism.

In [4], the authors propose temporal scopes as a language construct that can be used to specify the timing constraints of code execution and inter-process communication. A temporal scope allows the programmer to specify timing constraints and exception handlers to cope with timing errors. The language constructs introduced here, called resource scopes, extend temporal scopes with information about resource consumption and also specify semantics for the value domain.

6.1. Resource Scopes

In the most general form, a local resource scope allows the programmer to specify the timing of a statement block and its resource consumption. The resource scope's (approximate) form is as follows:

```
ts once::=
⟨bid⟩: start ⟨offset⟩ [ ⟨deadline⟩ ] [ ⟨vars⟩ ]
    [ ⟨resources⟩ ] do
    ⟨main body⟩ [ ⟨opt body⟩ ]
  done [⟨assign⟩]
  [donate ⟨don⟩]
```

The declaration of a scope starts with stating its identifier ⟨`id`⟩. The remainder of the construct's meaning is that the task is delayed as specified by ⟨`offset`⟩ and then its body, ⟨`main-body`⟩, and subsequently optional parts, ⟨`opt-body`⟩, are executed. The execution must complete before the ⟨`deadline`⟩. Within the body, it will access variables ⟨`vars`⟩ and will use the resources ⟨`resources`⟩. The construct terminates at its deadline, if provided; otherwise, the construct will continue endlessly until an error occurs. When the construct terminates, the assignments specified in ⟨`assign`⟩ are executed at the deadline.

If an error occurs, then the construct will be terminated and an exception will be raised. An error occurs, e.g., when the deadline is not met, the construct runs out of resources, or some illegal read/write operations happen. The raised exception is handled in the last active body's exception handler, if it is provided; otherwise, the exception is ignored and execution resumes with the next construct. The last active body, is either the body ⟨`st-body`⟩ or one of the optional ⟨`opt-body`⟩ ones. We discuss exception handling in Section 6.2.

The construct's offset can be specified using absolute time or relative time as follows:

```
⟨offset⟩::= now | at ⟨abs time⟩ | after ⟨rel time⟩
```

`now` means that there is no offset with the construct. `at` ⟨`abs-time`⟩ means that the body's execution is delayed until the specified physical time. `after` ⟨`rel-time`⟩ means that the body's execution's is delayed for ⟨`rel-time`⟩ time units.

The construct's deadline defines how soon the construct's execution has to be completed. As with the offset, a deadline can be specified using absolute time (`by` ⟨`abs-time`⟩) or relative time (`within` ⟨`rel-time`⟩) and its syntax is as follows:

```
⟨deadline⟩::= by ⟨abs time⟩ | within ⟨rel time⟩
```

The variables specifies which global variables are accessed within the construct. For each variable, the programmer must define its type and which global variable's value is used as initial value. The syntax is as follows:

```
⟨vars⟩::= with { ⟨pdef⟩ [, ⟨pdef⟩ ]* }
⟨pdef⟩::= ⟨type⟩ ⟨id⟩:=⟨value⟩
```

If declared, the variable list ⟨`vars`⟩ must contain at least one variable and each variable must have a unique identifier within the construct. The assigned value must have the same type as the declared variable. The declared variables are created immediately and the values are also assigned immediately. These semantics are discussed in detail in Section 6.2.

The resource declaration defines a collection of resources along with their type and amount. The resources are reserved at the construct's beginning and will dissipate when the construct terminates. Thereby, the construct consumes all resources specified in the collection. The syntax is as follows:

```
⟨resources⟩::= demand { ⟨rentry⟩ [, ⟨rentry⟩]+ }
⟨rentry⟩::= ⟨lbracket⟩ ⟨rtype⟩, ⟨numeral⟩ ⟨rbracket⟩
```

The resource entry, ⟨`rentry`⟩, specifies the resource type ⟨`rtype`⟩ (e.g., CPU, memory, communication), and the amount of resource as a numeral (e.g., CPU cycles, bytes of memory, and communication frames).

The main body is a statement block with a Java-like try-catch block around it:

```
⟨main body⟩::= ⟨block⟩ | try ⟨block⟩ ⟨catches⟩
⟨catches⟩::= ⟨catchclause⟩ | ⟨catches⟩ ⟨catchclause⟩
⟨catchclause⟩::= catch ( ⟨type⟩ ⟨id⟩ ) ⟨block⟩
```

The statement block ⟨`block`⟩ contains program instructions. The statement block uses the language specific to utilizing the resources. For example, it can use a C-like language for computation on the CPU. The try-catch block is similar to Java's one. The programmer can handle different exceptions independently by coding a separate catch-block. A construct can specify multiple optional bodies. The optional body will be executed only, if the scope has enough resources left after executing the main body. The syntax is as follows:

```
⟨opt body⟩::= optional ⟨ts once⟩
```

From the syntax, an optional body is a complete resource-scope construct, however, its semantics differ from the main body. We discuss the semantics and their difference in Section 6.2. The ⟨`assign`⟩ block is a simple list of assignment.

Finally, unused resources can be donated to other resource scopes:

```
⟨don⟩::= global | ⟨id⟩
```

The unused resources can be either donated to the global pool to be used by other optional scopes or they are specifically donated to one scope specified by ⟨`id`⟩.

```
 start now within 44ms
   demand { ⟨CPU,360K⟩, ⟨MEM, 110kB⟩ }
 do
   image_t my_image = acquire_image();

   optional start now within 3ms
     demand { ⟨CPU,10K⟩, ⟨MEM, 10kB⟩ }
     with { img := my_image }
   do
     img_status_t stat = img_sanity_check(img);
   done with ( cam_image_status:=stat )
 done with ( cam_image:=my_image );
donate global
```

Listing 1. The camera scope ($= RS_{cam}$).

Listing 1 shows the resource scope RS_{cam}, which implements the camera module. The module demands 360K instructions and 110kB of memory. The main body calls the method `acquire_image()` and acquires a fresh image from the camera. If resources are left, then RS_{cam} will perform a sanity check on the camera image. The resource scope terminates with copying the acquired image to the shared variable. Finally, the unused resources are donated to the global pool (e.g., unused CPU time and memory).

6.2. Semantics and Behavior

The previous paragraphs shows the resource scope's syntax; in the following, we present its behavior. We only provide an intuition and an informal specification of PEACOD's behavior, since the formal specification of the operational semantics of PEACOD are too long for this paper. At the moment, we have implemented an interpreter in Prolog to execute the scope and simulate its behavior. The interpreter can parses a resource scope construct and simulates the resource consumption, input/output behavior, and timing following a semantics description as described below. We plan to implement compiler for resource scopes.

6.2.1. PEACOD Behavior. A resource scope has semantics in three areas: the temporal domain, the value domain, and the resource domain. We use events to denote actions in the resource scope. The notation $ev_1 \leftrightarrow ev_2$ means that event ev_1 occurs iff event ev_2 occurs.

Definition 2 (Temporal Events)**.** *A resource scope σ can have the following temporal events:*

1) *The scope is released but is inactive (marked by event $ev_{rel}(\sigma)$).*
2) *The scope becomes active (event $ev_{act}(\sigma)$) after a specific offset $\mathrm{off}(\sigma)$ specified in the scope's temporal constraints, thus $ev_{act}(\sigma) \leftrightarrow ev_{rel}(\sigma) + \mathrm{off}(\sigma)$.*
3) *The scope finishes execution (event $ev_{fin}(\sigma)$) after a random amount of time has passed since the activation. This random mount of time can be described by execution-time distributions [8], hereafter labeled $\Upsilon(\sigma)$. Thus $ev_{fin}(\sigma) \leftrightarrow ev_{act}(\sigma) + \Upsilon(\sigma)$.*
4) *The scope completes (event $ev_{compt}(\sigma)$), i.e., all subscopes finished by (1) the transition system of σ reaches a final location, and (2) the scope at that location has finished. Thus, $ev_{compt}(\sigma) \leftrightarrow ev_{fin}(\sigma) + \sum_{\sigma_i} (\mathrm{off}(\sigma_i) + \Upsilon(\sigma_i))$ with σ_i all subscopes of σ that are activated in the run.*
5) *The scope terminates (event $ev_{term}(\sigma)$) when its deadline $dl(\sigma)$ passes: thus, $ev_{term}(\sigma) \leftrightarrow ev_{rel}(\sigma) + dl(\sigma)$.*

Definition 3 (Value Events)**.** *A scope σ can have the following value events where $\sigma.I$ marks the input variables in the $\langle$`vars`$\rangle$ section, sto_σ is the local variable storage, sto_G is the global variable storage, and $typ(v)$ refers to the declaration of variables as either* soft *or* hard.

1) *All hard input variables are read (event $ev_{I/hard}(\sigma)$): $\forall v \in \sigma.I \wedge typ(v) = hard : sto_\sigma.v \leftarrow sto_G.v$.*
2) *All hard output variables are written (event $ev_{O/hard}(\sigma)$): $\forall v \in \sigma.O \wedge typ(v) = hard : sto_G.v \leftarrow sto_\sigma.v$.*
3) *All soft input variables are read (event $ev_{I/soft}(\sigma)$): $\forall v \in \sigma.I \wedge typ(v) = soft : sto_\sigma.v \leftarrow sto_G.v$.*
4) *All soft output variables are written (event $ev_{O/soft}(\sigma)$): $\forall v \in \sigma.O \wedge typ(v) = soft : sto_G.v \leftarrow sto_\sigma.v$.*

A scope demands resources in the $\langle$`resources`$\rangle$ part of the declaration. We name the set of demands $\sigma.R$. We define *remaining demand* $\Delta(\sigma.R)$ as the allocated demand minus necessary demand to finish this particular execution. Note, that resources are allocated from a global pool. Optional scopes allocate their resources from their parent scope; however, optional scopes may use the resources from the global pool using. So in this sense, we have a hard reservation system but a soft consumption system (c.f., [9]).

Definition 4 (Resource Events)**.** *A scope σ can have the following resource events:*

1) *The scope allocates its demanded resources (event $ev_{alloc}(\sigma)$) according to $\sigma.R$.*
2) *The scope starts consuming allocated resources (event $ev_{consume}(\sigma)$).*
3) *The scope finishes consuming resources (event $ev_{noconsume}(\sigma)$). Note that preemption does not cause an event $ev_{noconsume}(\sigma)$, because a scope's demand has been calculated until the end of its execution. Thus $ev_{noconsume}(\sigma) \leftrightarrow ev_{consume}(\sigma) + \sigma.R - \Delta(\sigma.R)$.*
4) *The scope donates (event $ev_{donate}(\sigma)$) the remaining demand $\Delta(\sigma.R)$.*
5) *The scope frees resources (event $ev_{free}(\sigma)$).*

Definition 5 (PEACOD's Behavior)**.** *The behavior of a scope in PEACOD in the resource, value, and temporal domain is specified as:*

- *$ev_{alloc}(\sigma) \leftrightarrow ev_{rel}(\sigma) \leftrightarrow ev_{I/hard}(\sigma)$ meaning that a resource is allocated when the scope is released and when the hard variables are read.*
- *$ev_{consume}(\sigma) \leftrightarrow ev_a(\sigma) \leftrightarrow ev_{I/soft}(\sigma)$ meaning that a scope starts consuming resource when the offset has passed and when soft variables are read.*
- *$ev_{noconsume}(\sigma) \leftrightarrow ev_{fin}(\sigma) \leftrightarrow ev_{O/soft}(\sigma)$ meaning that a scope stops consuming resources when it finishes and when it writes to soft variables.*
- *$ev_{donate}(\sigma) \leftrightarrow ev_{compt}(\sigma)$ meaning that a scope donates resources when it completes.*
- *$ev_{free}(\sigma) \leftrightarrow ev_{term}(\sigma) \leftrightarrow ev_{O/hard}(\sigma)$ meaning that a scope frees its resources when its deadline passes and when it writes to hard variables.*

Additionally, we enforce the following access permissions:

- Required scopes must not read soft input values. Therefore, we check statically and dynamically that for each release event $ev_{rel}(\sigma) \wedge \sigma.\alpha = 1 : \nexists v \in \sigma.I : typ(v) = soft$.
- Optional scopes must not write hard output values. Therefore we check statically and dynamically that for each termination event $ev_{term}(\sigma) \wedge \sigma.\alpha < 1 : \nexists v \in \sigma.O : typ(v) = hard$.

6.2.2. Exception Handling. The $\langle$`main-body`$\rangle$ and $\langle$`opt-body`$\rangle$ can specify exception handlers. Possible exceptions cover value errors (e.g., invalid read/write access), resource errors (e.g., insufficient resources), and temporal errors (e.g., scope's release missed). If an error occurs, an exception is raised. While raising the exception, the system looses any kind of determinism. If a corresponding exception handler is available, then the system stops all active scopes to execute the exception handler. If no handler is available, the system ignores the exception and continues with the next construct. As the system handles the error, using a handler or ignoring the exception, at the scope's deadline, the remaining resources dissipate but no hard values will be written.

6.2.3. Compositional Determinism For Resource Scopes.

In the following, we now show that the resource scope construct with its specified semantics guarantees value–resource–time determinism. Note that we assume that the system executes resource scopes following the specification.

Theorem 3. *PEACOD's behavior guarantees value–resource–time determinism, i.e.,* $\forall tr, tr' : tr \sim_v tr'$ *with* $v = \{clk, I, O, cc\}$.

Proof: We proceed to prove Theorem 3 by contradiction. For that we have to show that $\exists tr, tr' : tr \not\sim_v tr'$ with $v = \{clk, I, O, cc\}$. This is the case, if either tr or tr' contain a task state, that is not present in the other's run. Let us consider this on a case-by-case basis:

- *Task state changes w.r.t. clk.* Scopes always terminate (run out of time, run out of resources) and therefore their runs always have a beginning and an ending. For two runs tr and tr' to ensure $tr \not\sim_{\{clk\}} tr'$, either the beginning or the ending has to differ between runs. The two events that affect this are ev_{rel} and ev_{term}. In PEACOD, the occurrence of both events is invariant of the run time (Υ) and thus it is impossible to create two runs, where the occurrence of these two events differs w.r.t. time. Therefore, $\forall tr, tr' : tr \sim_{\{clk\}} tr'$.
- *Task state changes w.r.t.* I, O. Runs change whenever output variables are written or inputs are read. For two runs tr and tr' to ensure $tr \not\sim_{\{clk,I,O\}} tr'$, one of the runs must read or write a variable while the other does not. The events $ev_{I/hard}$ and $ev_{O/hard}$ control reading and writing result in task state changes. Again, in PEACOD these two events are invariant of the run time, thus it is also impossible to create two runs, where their occurrence differs w.r.t. time. Thus, $\forall tr, tr' : tr \sim_{\{clk,I,O\}} tr'$
- *Task state changes w.r.t. cc.* Runs change whenever the consumption configuration changes. For two runs tr and tr' to ensure $tr \not\sim_{\{clk,cc\}} tr'$, one of the runs must change the consumption configuration at a different time than the other task. Changes in the consumption configuration occur through the events ev_{alloc} and ev_{free}. Similar to above, in PEACOD it is impossible to create two runs, which are dissimilar w.r.t. time and consumption. Therefore, $\forall tr, tr' : tr \sim_{\{clk,cc\}} tr'$.

Since it is impossible to generate two runs tr and tr' that satisfy $tr \not\sim_{\{clk,I,O,cc\}} tr'$, Theorem 3 follows. □

Corollary 1. *PEACOD always produces the same run and thereby guarantees:* $\forall tr, tr' : tr = tr'$.

Proof: Based on Theorem 3, it is sufficient to show that: $tr = tr' \Leftrightarrow tr \sim_{\{clk,I,O,cc\}} tr'$.

It is trivial that $(tr = tr') \implies tr \sim_{\{clk,I,O,cc\}} tr'$. $tr \sim_v tr'$ states that all state changes are the same in both runs according to the view v. If the view includes all elements ($v = \{clk, I, O, cc\}$), then $tr \sim_v tr' \implies tr = tr'$ in all elements. □

Compositional determinism requires that the task properties concerning determinism are preserved: i.e., if τ_1 and τ_2 are value-time deterministic, then $\tau_3 = (\tau_1 \| \tau_2)$ also to be value-time deterministic assuming that τ_3 is both schedulable and results in no race condition.

A race condition can occur for $\tau_3 = (\tau_1 \| \tau_2)$ if the following conditions hold: (a) τ_1 and τ_2 share an output variable, (b) τ_1 and τ_2 allocate the same mutually exclusive resource, or (c) τ_1 and τ_2 allocate the same time-shared resource. The first two cases prevent us from composing the system, since we cannot resolve the race condition. In the last case, we can first check whether the combined demand $(\tau_1 \| \tau_2)$ remains schedulable (see Section 5) and then combine the demands of τ_1 and τ_2 into a single allocation request in τ_3.

Theorem 4. *PEACOD provides compositional determinism with respect to time, values, and resources (*$Z = \{clk, I, O, cc\}$*) assuming that the composed system remains schedulable, resource scopes share no output variables, and mutually exclusively allocate mutually exclusive resources..*

Proof: We can define $\tau_3 = \tau_1 || \tau_2 = \langle clk, \mathcal{V}_i(\tau_1) \cup \mathcal{V}_i(\tau_2), \mathcal{V}_o(\tau_1) \cup \mathcal{V}_o(\tau_2), cc(\tau_1) \cup cc(\tau_2) \rangle$. Now the result follows from Theorem 1. □

Note that PEACOD also supports sequential composition $(\tau_1; \tau_2)$. We can combine sequentially composed resource scopes by calculating one resource scope that treats the scopes are sub-scopes. A race condition can occur for $\tau_3 = (\tau_1; \tau_2)$ if the following conditions holds: (a) τ_1 has an output variable that is an input variable of τ_2 or (b) τ_1 releases a mutually exclusive resource that is allocated by τ_2. In both cases, we resolve the possible race condition by first completing τ_1—writing its variables, releasing resources—and then executing τ_2.

7. Implementation Outline

The PEACOD framework is, to date, a formal model with specified semantics, an interpreter in Prolog, and proven properties. However, in this section, we provide a detailed outline of how PEACOD can be implemented and why we think it is feasible. The central argument is that related systems have already implemented one or the other type of determinism and PEACOD can utilize and extend these mechanisms.

The centerpiece of the implementation is a resource, value, and timing mediator, following the mediator design pattern [10]. A mediator defines an object that encapsulates how a set of objects interact. The mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

The *timing mediator* controls the temporal domain and coordinates releasing, activating, completing, and terminating scopes. Timing mediators have already been successfully implement in a variety of real-time operating systems [9], e.g., schedulers that support periodic tasks. The *value mediator* controls the value domain and coordinates variable read and write access for each scope. A value mediator has already been successfully implemented by the Giotto [3], achieving value/time determinism. A central element, the E machine that controls write and write access to values via, what they call, drivers. Finally, the *resource mediator* controls the resource domain and coordinates allocation, donations, and dissipation. Resource reservation-based systems already provide a resource mediator (see nano-RK [11] or other resource kernels in [9]). PEACOD's donation system can then be implemented on top of the resource reservation system.

Without a tool chain, programming resource scopes is a cumbersome task. Parts of the tool chain already exist as individual products or research prototypes. The tool chain should provide resource-consumption data and should support code generation. Determining network consumption is not uncommon as a number of real-time protocols require this information to calculate the bus schedule. Tools to determine the worst-case execution time are already available and usable [12]. Algorithms for worst-case memory consumption are in their early stages [13]. So, in conclusion, the individual pieces do exist but have to be brought together and integrated into one tool chain.

8. Related Work

Temporal scopes [4] allow the programmer to specify timing constraints and exception handlers to cope with timing errors. This work's goals are similar and build upon the work of temporal scopes. In this work, we extend temporal scopes to allow the programmer to specify time and resource constraints and also provide semantics for value and resource usage.

A number of systems provide determinism in one or two domains. Programmable logic controllers (PLCs) are deterministic in the resource and value domain. Before computing, a PLC creates a process image and all computation bases on this image. At the finish time, the all computed values are written. Giotto [3] implements value/-time determinism by enforcing reading and writing variables are specific points in time. Some resource kernels and real-time operating systems such as nano-RK [11] provide resource/time determinism w.r.t. individual resources. For example nano-RK implements resource/time determinism for sensors, since the sensor is allocated at the release time and released at the finish time. In contrast to this work, PEACOD introduces resource/time/value determinism. Furthermore, PEACOD provides hard and soft values to handle values with low latency (as specified, soft values are written before the deadline, however, they are not deterministic; other system do not offer this dualism) and lowers the system overhead by its resource donation system.

A number of programming languages are common in the area of embedded systems such as Ada and C. Some programming languages also provide specific constructs for real-time systems such as the real-time specification for Java (RTSJ) [14] or the time-triggered message-triggered objects (TMO) [15]. Out of these and related languages, some provide constructs for timing such as a delay statement in Ada and C or similar to temporal scopes, textual specifications in TMO. Some programming languages provide constructs to control resources such a RTSJ with the `ScopedMemory` object for memory and the object `SchedulingParameters` for CPU consumption. However, these languages do not provide value–time and resource–time determinism together, which is essential. This work differs as it provides these two types of determinism and provides a compositional non-growing abstraction.

In their work, I. Shin et al [1] defined a composition abstraction for a periodic task and resource model. This abstraction preserves the task's semantics and permits composing tasks under EDF and RM. In this work, we provide an abstraction for a more general task model with offsets and deadlines different from the period. This general task model allows us to decouple resource consumption and together with the compositional abstraction, it permits compositionality in the value and resource domain. Others define composition via arrival curves [2], which covers a general task model, but ignores value behavior.

In resource-aware programming [16] a users can monitor the resources used by their programs and programmatically express policies for the management of such resources. The framework bases on a notion of hierarchical groups, which act as resource containers for the computations they sponsor. However, the main difference is that the resource initially allocated can be exhausted or retrieved if not consumed. Therefore, the resource consumption cannot be determined a priori and the user interaction is needed to handle these cases. Our approach, in contrast, aims at providing deterministic guarantees on resource consumption.

Determinism, as defined by Kopetz [17] represents a specific form of determinism that is compatible and can be expressed with our notion of determinism; in fact, Kopetz's model is value–time deterministic as per our definition.

9. Conclusions

Different models and systems discuss or enable determinism in one or two of the domains resource, value, and time to model or support compositionality. Each of them makes assumption about the third, missing domain such as ignoring value behavior or assuming unlimited resources.

Consequently, realizing a physical system with these models can cause problems as these assumptions may be violated.

The PEACOD framework provides one model that supports deterministic behavior in all three domains and allows compositionality under this unified view. In this work, we presented the framework and its language construct. We introduced the necessary terminology to define determinism in the time, value, and resource domain on the basis of similar runs and proved that our specified behavior for resource scopes in PEACOD guarantees deterministic behavior in all three domains. Furthermore, we proved that it preserves this property under composition.

By providing both determinism for these domains and compositionality, the PEACOD framework hopes to ease the design and implementation of real-time embedded systems. In the future, we hope to relax some of the rigid assumptions made in this work, towards making this framework more practical.

Firstly, we plan on using the conditional task model [18] to model the resource scopes. Conditional task models are more expressive than the periodic task models used here, and the results on compositional analysis with conditional task model can be used to synthesize a tighter abstraction for all the system tasks.

Secondly, we hope to use resource monitoring to fill in the resource requirements of each scope, instead of having to specify them explicitly. Additionally, we would also be looking at reservable and schedulable resources, and implementing the resource, value, and timing mediator to allow experimenting with the framework.

Finally, we also plan on extending the model to include faults and exceptions. This would require defining deterministic behavior in the presence of faults, and methods to realize them in the implementation.

References

[1] I. Shin and I. Lee, "Periodic Resource Model for Compositional Real-Time Guarantees," in *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[2] E. Wandeler and L. Thiele, "Real-Time Interfaces for Interface-Based Design of Real-Time Systems with Fixed Priority Scheduling," in *Proc. of the ACM Conference on Embedded Software (EMSOFT'05)*, pp. 80–89, 2005.

[3] T. A. Henzinger, C. M. Kirsch, and B. Horowitz, "Giotto: A Time-triggered Language for Embedded Programming," in *Proc. of the 1st International Workshop on Embedded Software (EMSOFT)* (T. A. Henzinger and C. M. Kirsch, eds.), no. 2211 in LNCS, Springer, Oct. 2001.

[4] I. Lee and V. Gehlot, "Language Constructs for Distributed Real-Time Programming," in *Proc. of the IEEE Real-time Systems Symposium (RTSS)*, 1985.

[5] G. Buchman, D. Cohen, P. Vernaza, and D. Lee, "The University of Pennsylvania Robocup 2005 Legged Soccer Team," 2005.

[6] J.-T. Leung and M. Merrill, "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks," *Information Processing Letters*, vol. 11, no. 3, pp. 115–118, 1980.

[7] S. Baruah, A. Mok, and L. Rosier, "Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor," *Real-Time Systems Journal*, vol. 2, pp. 301–324, 1990.

[8] M. L. Li, T. V. Achteren, E. Brockmeyer, and F. Catthoor, "Statistical Performance Analysis and Estimation of Coarse Grain Parallel Multimedia Processing System," in *RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, (Washington, DC, USA), pp. 277–288, IEEE Computer Society, 2006.

[9] J. A. Stankovic and R. Rajkumar, "Real-Time Operating Systems," *Real-Time Systems*, vol. 28, pp. 237–253, 2004.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[11] A. Eswaran, A. Rowe, and R. Rajkumar, "nano-RK: An Energy-Aware Resource-Centric Operating System for Sensor Networks," in *Proc. of the IEEE Real-Time Systems Symposium (RTSS'05)*, Dec. 2005.

[12] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying Static WCET Analysis to Automotive Communication Software," in *Proc. of the 17th Euromicro Conference on Real-Time Systems (ECRTS)*, (Mallorca, Spain), July 2005.

[13] P. Persson, "Live memory analysis for garbage collection in embedded systems," in *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems (LCTES)*, (New York, NY, USA), pp. 45–54, ACM Press, 1999.

[14] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[15] K. Kim, "Object Structures for Real-Time Systems and Simulators," *IEEE Computer*, vol. 30, no. 8, pp. 62–70, 1997.

[16] L. Moreau and C. Queinnec, "Resource-Aware Programming," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 441–476, 2005.

[17] H. Kopetz, "The Rationale for Time-Triggered Ethernet," *Real-Time Systems Symposium, 2008*, pp. 3–11, 30 2008-Dec. 3 2008.

[18] M. Anand, A. Easwaran, S. Fischmeister, and I. Lee, "Compositional Feasibility Analysis for Conditional Task Models," in *Proceedings of the Eleventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'08)*, (Washington, DC, USA), IEEE Computer Society, 2008.