Formal Verification of Distributed Preemptive Real-time Scheduling

Gabor Madl[1,2] Sherif Abdelwahed[1]

[1]Institute for Software Integrated Systems,
Vanderbilt University, Nashville, TN 37205

[2]Center for Embedded Computer Systems,
University of California, Irvine CA 92697

Abstract

The application of component middleware in mission-critical systems introduces new challenges for distributed real-time embedded (DRE) system developers such as the safe composition of components to ensure end-to-end predictability. Model-based analysis provides a way to evaluate design alternatives with respect to functional specifications and the target platform. This paper introduces a semantic domain which captures key time-based properties of a generic class of DRE systems. We utilize this semantic domain to verify the preemptive schedulability using existing model checking tools. The proposed verification method and framework is demonstrated on a mission-critical avionics DRE system.

1. Introduction

Distributed real-time embedded (DRE) systems, including command and control systems, wireless sensor networks and avionics mission computing, increasingly run in open environments, under highly unpredictable conditions. As a result, current and planned DRE systems require a highly adaptive and flexible infrastructure that can factor out reusable resource management services from application code, and provide safe methods for system synthesis.

When dealing with multiple Quality of Service (QoS) properties, DRE system designers often face a gap between design and implementation, since properties are specified in declarative way such as worst case execution times, priorities or in even broader range such as adaptible, fault-tolerant etc. Contrary, implementations typically follow an imperative approach which makes hard to reconstruct system structure and behavior from the source code. To address these challenges, it is useful to analyze system behavior at design-time, thereby enabling developers to select suitable design alternatives before committing to specific platforms.

Several design methodologies were introduced for embedded systems' design which provide effective methods and frameworks to bridge the gap of mapping functional specifications to the target platform. Platform-based design [1], abstract semantics [4, 22], model-based design [19] and pattern-oriented design [30] are all candidates which have proven their effectiveness in various application domains.

Model-based analysis techniques can provide high-level abstractions that capture crucial concepts of the system operation such as structure, behavior, environment and the properties it must satisfy. We emphasize that models are essential in the design, configuration, integration and optimization and in our opinion should be essential in the analysis as well. *Model-based verification* techniques [6,11] provide a way for the design-time analysis of many engineering systems including DRE systems and allow, through abstraction, to overcome accidental complexities associated with third generation programming languages such as memory management and pointers. Furthermore, they provide a way to observe and prove properties specified in a declarative way by analyzing the composition of imperative component implementations.

This paper utilizes the DRE Semantic Domain 4 as a formal basis for the analysis of distributed sensor network applications. The verification algorithm checks end-to-end deadlines and latencies for fixed-priority scheduling. This method captures task dependencies, priorities and delays in the communication and execution intervals as well. The analysis utilizes a novel conservative approximation scheme to verify the *preemptive* scheduling of distributed multiprocess real-time

embedded systems. The validity of the approach can be verified by available model checkers such as UPPAAL [26, 21], Kronos [7] or the IF [2] toolset.

The structure of the paper is the following: Section 2 introduces the problems to be solved; Section 3 explains how model-based verification can be applied to provide a way for the design-time analysis of DRE systems; Section 4 introduces the semantic domain used for the analysis; Section 5 gives a detailed example for the verification; Section 6 compares our work on model-based verification with related work; and Section 7 presents concluding remarks.

2. Problem Formulation

This paper considers the problem of deciding the schedulability of a given set of tasks with event- *and* time-driven interactions, on a distributed preemptive platform.

Definition 1 *The system is schedulable if all tasks finish their execution before their respective deadlines.* □

We use a timed automata formulation of the problem which translates the schedulability problem into a reachability problem in which the set of tasks are schedulable if a predefined `error` state is not reachable in any of the tasks' timed automata. If this analysis completes successfully it implies that all tasks complete before their respective deadlines.

The proposed model of computation corresponds to the stopwatch model due to assignments of variables to clocks. The stopwatch model can be expressed using hybrid automata, where in the preempted state the local clock keeps a constant value ($\dot{x} = 0$), otherwise it progresses linearly ($\dot{x} = k, k \in \mathbb{N}^+$).

Deciding the preemptive schedulability of the stopwatch model has been shown to be undecidable using timed automata [20]. To address this issue, we implement a discretized preemption scheme in which task interruption can only be granted at specific intervals during the task execution. This method gives an over-approximation of the safe states of the system, when the schedulable discretized approximation implies a schedulable system in continuous time. By using the timed automata approximation scheme we can achieve superior performance in the verification compared to hybrid automata verification.

Definition 2 *We define a frame period for a task $t \in T$, referred to as* $\mathsf{Period}(t)$ *as the period of the slowest timer on which the task depends.* □

Tasks that are assigned to the same platform processor and have the same frame period are in the same frame. Therefore, the set of frames is a partition of the set of tasks T. For a task t we write $\mathsf{Frame}(t)$ to identify the set of all tasks in T belonging to the frame of t.

Theorem 1 *If the system is schedulable using discretized time with preemptions, it is also schedulable in continuous time if*

$$(\forall t \in T) \quad \mathsf{D}(t) \geq \mathsf{Period}(t) - \sum_{t' \in \mathsf{Frame}(t)-t} \mathtt{WCET}(t')$$

where $\mathsf{D}(t)$ *denotes the deadline of the task and* $\mathtt{WCET}(t)$ *denotes the worst case execution time of the task.*

Proof (Outline) Suppose on the contrary there exists a schedule in discretized time which is not schedulable in continuous time. This implies from Definition 1 that all tasks finish their execution before their respective deadlines in discrete time, whereas some tasks do not finish their respective deadlines in continuous time. Since none of the tasks spend less time executing this implies that some tasks spend less time either in the preempted or enabled state. A task can only spend less time in the preempted state if the preempting task finishes the execution faster which is a contradiction.

Therefore, the only scenario possible is when a task waits initially for shorter time than in the continuous time. This can only happen when the task receives the start event at some later time instance. This implies that it won't finish the execution earlier than in the continuous case, only we delay with counting the deadline thus making an unschedulable task schedulable. This phenomenon is the result of the deadline semantics which define deadlines respective to being enabled rather than respective to the timer which triggered the computation.

By ensuring that deadlines are constrained by as defined in the theorem we ensure that a schedulable system in discrete time implies a schedulable system in the continuous time as well, since even if the action has to wait for all tasks before starting its execution it will finish before the end of the frame as in Definition 2. □

3. Model-Based Verification

Designing a DRE application which satisfies multiple QoS properties is a complex constraint-satisfaction problem. To ensure optimal QoS support in practical applications developers often need to face hard or even undecidable problems. Despite recent advances in embedded and hybrid systems' analysis [22, 3, 2, 18] and abstraction techniques [15] the generic verification of

industry-size DRE systems is largely unsolved. Time-triggered systems offer better analyzability [14,28] but increase the implementation complexity and resource needs to satisfy the synchronization.

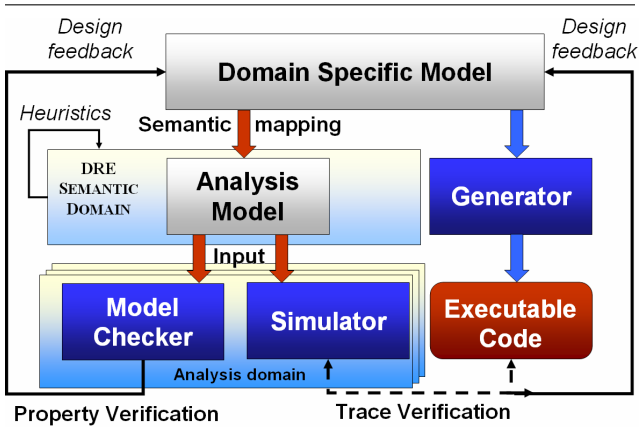

Figure 1. Model Checking with Heuristiscs

Figure 1 shows the proposed method for the real-time analysis of distributed embedded systems. This method builds on the results of platform-based design [1] and model integrated computing (MIC) [32]. Domain-specific modeling languages (DSMLs) play an essential role in the design and analysis process. The analysis follows the model checking approach [10] to prove QoS properties of DRE systems.

Designing an application that satisfies multiple QoS properties is a multi-step process in which the domain-specific model is continually evolved until the underlying method verifies that the QoS properties are satisfied. This evolution is performed by DRE system designers based on the feedback from the model checking tool.

The goal of the analysis is to aid system development by choosing feasible design alternatives. Heuristics play an important role in this approach by applying key constraints to the analysis model which can be checked effectively. The result of the checks can be reported back to the DRE system developer or can be used to change the domain specific models automatically. For example, heuristics can be used to obtain priority assignments for tasks such that the system is schedulable.

When designing applications with multiple QoS support the analysis often requires multiple tools for the analysis. For example, one tool can be used to verify real-time properties while a simulation can be used to predict the overall power consumption of the system. The DRE Semantic Domain provides a common semantic domain which can capture multiple QoS properties of a generic class of DRE systems, including sensor networks. The common semantic domain provides the basis for the analysis of the domain-specific model.

Simulators are often integrated into the analysis tools and use the same model of computation. When the model checking tool finds an unsatisfied property, such as system deadlock (*e.g.*, it may find a deadlock when checking the formal model of computation generated from the domain-specific application model) the simulator can be used to simulate the execution trace that yields this undesired behavior.

DSMLs are used to synthesize executable code, simulations or documentation using the Model Integrated Program Synthesis (MIPS) [33]. Since the DSML application models are semantically linked to the formal models of computation, the simulator and the generator must produce the same execution traces for the same input data.

We use the DRE Semantic Domain as a common semantic domain for the analyis. This domain has been semantically anchored to the timed automata model of computation and can be checked by model checkers which use timed automata as an underlying model of computation such as Uppaal [26,21], Kronos [7] or the IF [2] toolset. However, as our results show a formal proof of correctness can be obtained by applying the verification method explained in Section 4.

4. The DRE Semantic Domain

In this section we formalize a computational model that can express the event-driven nature of DRE systems. We define a model on a distributed platform with possible execution preemptions. The proposed model of computation is a composition of generic dataflow, which is suitable to express event-driven communication and timed finite state machines. We refer to this semantic domain as DRE Semantic Domain.

- The DRE Semantic Domain allows the specification of task dependencies and the mapping of tasks to a distributed multi-threaded, multi-processor platform and therefore is a generic to most DRE system.
- The DRE Semantic Domain incorporates a real-time event channel component, which allows (1) modeling of asynchronous non-blocking communication between tasks, (2) precise modeling of the buffering in real-time event channel threads, (3) capturing the exact delay of event propagations on the event channel, (4) timers to be deployed to

different processors than the tasks which are subscribed to them.

The DRE SEMANTIC DOMAIN captures the behavior of event-driven DRE systems. To facilitate formal developments, and without losing generality, we assume that communications through shared variables or common coupling are not considered in the model and delays within a processor are ignored since such delays are orders of magnitude less than the delays incurred by the real-time event channels between hosts.

4.1. Syntax

The DRE model of computation is a tuple $M = \{T, C, TR, PR\}$ where:

- T is a set of *tasks*,
- C is a set of *real-time event channels*,
- TR is a set of *timers*, which are special tasks that publish events at a given rate.
- PR is a set of *platform processors*.

Tasks and timers are assigned to execute on a specific processor. The processor associated with a given task or timer is specified by the map $\mathsf{Processor} : T \cup TR \rightarrow PR$. Timers generate periodic events as specified by the map $\mathsf{Period} : TR \rightarrow \mathbb{N}^+$.

Tasks are attributed by their priorities, sub-priorities, deadlines, and worst-case execution times. These properties are specified by the maps, P, SP, D, WCET, respectively. In addition, tasks are dependents on triggering tasks, timers or event channels. Event channels are attributed by the buffer size and the worst case delay of delivering published events. These properties are specified by the B and δ maps, respectively. Connections inside the computation model are defined by the TaskToTask, TaskToChannel, ChannelToTask, TimerToChannel and TimerToTask functions which return the corresponding elements to which the current element is connected.

4.2. Semantics

Tasks are executed as soon as they are enabled by system events, they are the highest sub-priority task in the priority group and no other higher priority task is enabled. Tasks can be preempted during execution by higher priority tasks and will resume operation as soon as all enabled higher priority tasks completed their current execution cycle and are no longer enabled.

4.2.1. Time and Clocks In the continuous time model, system clocks (global and local) are assigned a value in $\mathbb{R}^+$ the set of non-negative real numbers. For a given clock assignment x, we write x' to denote the immediate next clock assignment. In the DRE SEMANTIC DOMAIN, execution of tasks is coordinated with respect to a global clock. The current global clock value is denoted by x_g. We associate with each timer $r \in TR$ a local clock x_r that indicates the time remaining in the current cycle. We assume that timers are all triggered at the start of execution $x_g = 0$. We associate with each task t a local clock x_t that indicates the time spent in the execution state in the current execution cycle.

4.2.2. System State The state of the system is defined as the composite state of its tasks, event channels and timers.

For a dynamic element a, we write $\mathsf{State}(a, x)$ for the value of a at (global) time x. We now define the state domain of each model elements.

- $(\forall t \in T)(\forall x \in \mathbb{R}^+)\ \mathsf{State}(t, x) \in \{\texttt{idle}, \texttt{enabled}, \texttt{executing}, \texttt{preempted}\}$
- $(\forall c \in C)(\forall x \in \mathbb{R}^+)\ \mathsf{State}(c, x) \in \{\texttt{idle}, \texttt{send}, \texttt{receive}, \texttt{buffer}, \texttt{wait}\}$
- $(\forall r \in TR)(\forall x \in \mathbb{R}^+)\ \mathsf{State}(r, x) = \mathrm{mod}(x, \mathsf{Period}(r))$, where the function $\mathrm{mod}(x, y)$ returns the remainder of dividing x by y.

For a given computation model $M = \{T, C, TR, PR\}$, the state of the overall DRE model is given by:

$$\mathsf{State}(M, x) = \prod_{a \in T \cup C \cup TR} \mathsf{State}(a, x)$$

Note that the state of every system element is well-defined at any time x and so does the composite state of the system.

4.2.3. System Events System events corresponds to changes in the state of one or more of the dynamic components of the model. We denote the set of events as E. Some events are associated with a set of types. This association is defined by the partial map $\mathsf{type} : E \rightarrow \{\texttt{start}, \texttt{finished}\}$.

We introduce the functions $\mathsf{send}(a, e, x)$ and $\mathsf{receive}(a, e, x)$. If a task a sends out an event e at (global) time x by $\mathsf{send}(a, e, x)$ then $\mathsf{receive}(b, e, x')$ evaluates to `true` for a dynamic element b, otherwise it evaluates to false.

Events are broadcasted to dynamic elements that are connected to the generating element. These connections are defined by the TaskToTask, TaskToChannel, ChannelToTask, TimerToChannel, TimerToTask

maps, respectively, and follow a non-blocking semantics, namely, if they are not received they get lost.

4.2.4. System Transitions and Trajectories Systems transitions correspond to valid state changes and the associated events. Validity of transitions are established by the following rules. A trajectory is a sequence of valid state changes and the corresponding triggering events. The set of all trajectories corresponding to a set of state variables defines the behavior of the underlying dynamic elements.

Timers generate events periodically, namely when the corresponding state (local clock) evaluates to 0, according to the following rule:

$(\exists r \in TR)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ \mathsf{State}(r, x) = 0 \ \rightarrow \mathsf{send}(r, e, x')\ \wedge\ \mathsf{type}(e) = \texttt{start}$

Tasks are cyclic execution units that change their state based on a preemptive scheduling policy as described earlier. Tasks are initialized at an `idle` state, that is, $\mathsf{State}(t, 0) = \texttt{idle}$, for all $t \in T$. An idle task will be enabled when it receives an event:

$(\forall t \in T)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(t, x) = \texttt{idle} \wedge \mathsf{receive}(t, e, x)\ \wedge\ \mathsf{type}(e) = \texttt{start}) \ \rightarrow\ \mathsf{State}(t, x') = \texttt{enabled}$

A task will be executed if it is enabled and no other higher priority or sub-priority task is enabled on the same processor. To simplify notation we write T_S for the set of tasks assigned to the processor S. Then for a task $t \in T$ with $S = \mathsf{Processor}(t)$ we have

$(\forall t \in T)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(t, x) \in \{\texttt{enabled}, \texttt{preempted}\}\ \wedge\ (\forall t' \in T_S)\ (\mathsf{P}(t') > \mathsf{P}(t)\ \vee\ \mathsf{SP}(t') > \mathsf{SP}(t)) \ \rightarrow\ \mathsf{State}(t', x) = \texttt{idle}) \ \rightarrow\ \mathsf{State}(t, x') = \texttt{executing}$

A task will be preempted if it is executing and another higher priority task becomes enabled on the same processor. That is, for a task $t \in T$ with $S = \mathsf{Processor}(t)$ we have

$(\forall t \in T)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(t, x) = \texttt{executing}\ \wedge\ (\exists t' \in T_S)(\mathsf{P}(t') > \mathsf{P}(t) \ \rightarrow\ \mathsf{State}(t', x) = \texttt{enabled})) \ \rightarrow$
$\mathsf{State}(t, x') = \texttt{preempted}$

A task t will return to the `idle` state if it is executing and its internal clock evaluates to $\mathsf{WCET}(t)$:

$(\forall t \in T)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(t, x) = \texttt{executing}\ \wedge\ x = \mathsf{WCET}(t)) \ \rightarrow\ \mathsf{State}(t, x') = \texttt{idle}\ \wedge\ \mathsf{send}(t, e, x')\ \wedge\ \mathsf{type}(e) = \texttt{finished}$

Generating an event at the end of the execution is optional. An event channel receives events from a timer or task and propagates it to a task with some delay δ. If the task is in an execution cycle (not in the `idle` state) it buffers the events. We formalize state transitions in the event channels next. When the task receives a start event it either propagates it or buffers it according to the state of the subscribed task. $buffer_c$ represents the buffer:

$(\forall c \in C)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{idle} \wedge \mathsf{receive}(c, e, x)\ \wedge\ \mathsf{type}(e) = \texttt{start}) \ \rightarrow\ (\mathsf{State}(c, x') = \texttt{receive}\ \wedge\ buffer_c++)$

$(\forall c \in C)(\forall t \in T)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{receive}\ \wedge\ \mathsf{ChannelToTask}(c) = t\ \wedge\ \mathsf{State}(t, x) = \texttt{idle}) \ \rightarrow\ \mathsf{State}(c, x') = \texttt{wait}$

$(\forall c \in C)(\forall t \in T)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{receive}\ \wedge\ \mathsf{ChannelToTask}(c) = t\ \wedge\ \mathsf{State}(t, x) \neq \texttt{idle}) \ \rightarrow\ \mathsf{State}(c, x') = \texttt{idle}$

When a subscribed task finishes the execution cycle the event channel moves to the `wait` state if the buffer is not empty:

$(\forall c \in C)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{idle} \wedge \mathsf{receive}(c, e, x)\ \wedge\ \mathsf{type}(e) = \texttt{finished}) \ \rightarrow\ \mathsf{State}(c, x') = \texttt{send}$

$(\forall c \in C)(\forall t \in T)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{send}\ \wedge\ buffer_c = 0) \ \rightarrow\ \mathsf{State}(c, x') = \texttt{idle}$

$(\forall c \in C)(\forall t \in T)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{send}\ \wedge\ buffer_c < \lambda_c) \ \rightarrow\ \mathsf{State}(c, x') = \texttt{wait}$

After the delay δ has passed the event channel generates an event for the subscribed task if the subscribed task is idle:

$(\forall c \in C)(\forall t \in T)(\forall x \in \mathbb{R}^+)(\forall y \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{wait}\ \wedge\ \mathsf{ChannelToTask}(c) = t\ \wedge\ \mathsf{State}(t, x) = \texttt{idle} \ \rightarrow\ (\mathsf{send}(c, e, y)\ \wedge\ \mathsf{type}(e) = \texttt{start}\ \wedge\ buffer_c--\ \wedge\ \mathsf{State}(c, y) = \texttt{idle}\ \wedge\ y - x < \delta)$

$(\forall c \in C)(\forall t \in T)(\forall x \in \mathbb{R}^+)(\forall y \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{wait}\ \wedge\ \mathsf{ChannelToTask}(c) = t\ \wedge\ \mathsf{State}(t, x) \neq \texttt{idle} \ \rightarrow\ \mathsf{State}(c, y) = \texttt{idle}\ \wedge\ y - x < \delta)$

When an event is received in the waiting state the event channel buffers it:

$(\forall c \in C)(\forall t \in T)(\forall e \in E)(\forall x \in \mathbb{R}^+)\ (\mathsf{State}(c, x) = \texttt{wait}\ \wedge\ \mathsf{receive}(c, e, x)\ \wedge\ \mathsf{type}(e) = \texttt{start}) \ \rightarrow\ buffer_c++$

4.3. Timed Automata Models

The DRE Semantic Domain can be modeled by a set of timers, dynamic computation tasks, event channels and schedulers. DRE system models can be built by the composition of these components. First we introduce the components used for the modeling and then we define the scheduling of the overall model.

4.3.1. The Timer The timer is a simple periodic event generator which releases task initiation events at a specified rate. Timers may represent sensors sampled at a predefined rate and can arbitrarily drift from each other. However, we assume that time drifts in timers are bounded. Timers can be represented by a generic timed automaton model shown in Figure 2.

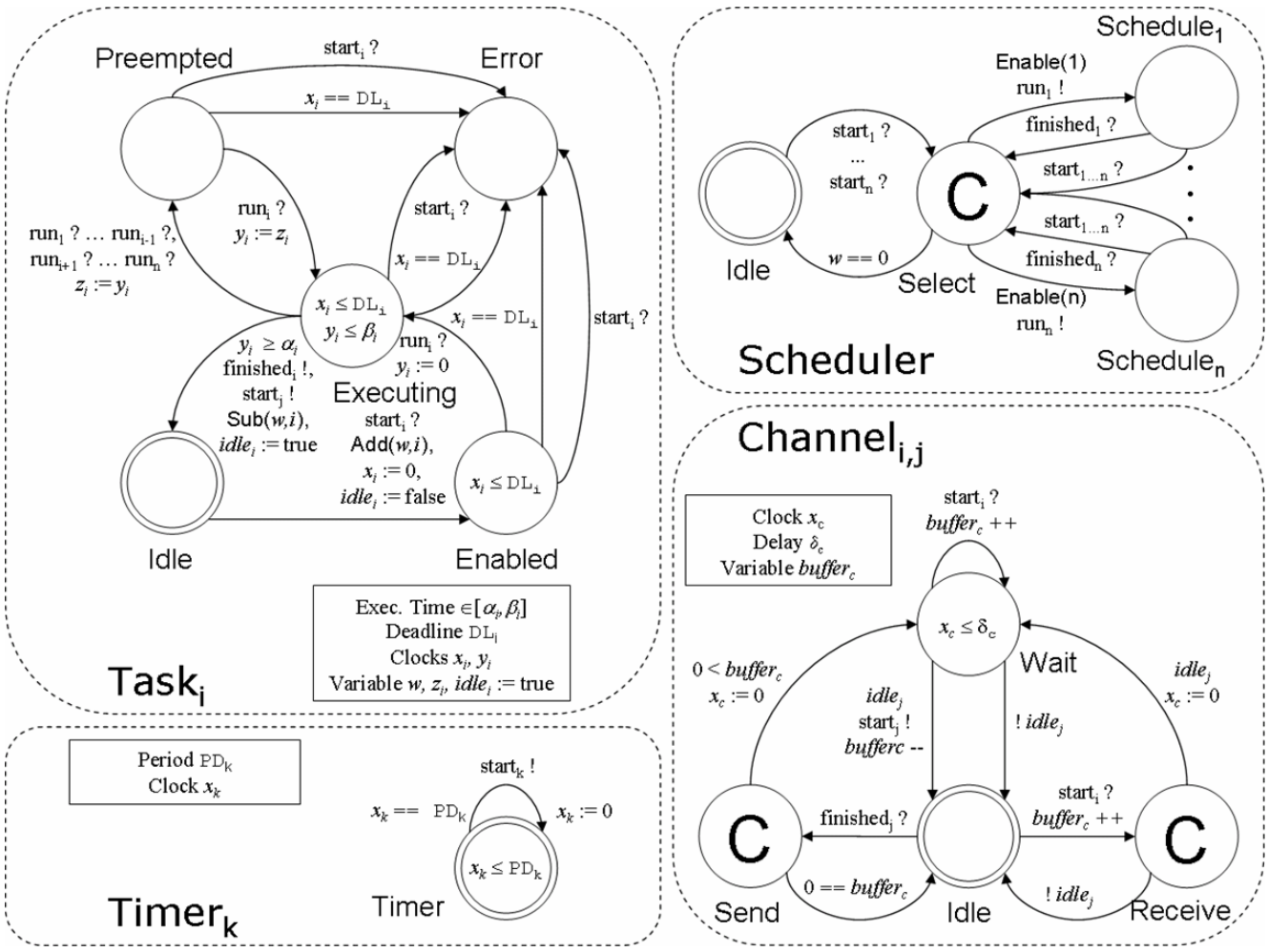

Figure 2. Generic Model of the DRE Semantic Domain

4.3.2. The Task Tasks are the main components which describe the actors in DRE systems. In addition to the four states defined in Section 4 (`idle`, `enabled`, `executing` and `preempted`), we also introduce a `timeout` state which can be used to express undesired behavior of the system such as when the task cannot finish the execution before its respective deadline.

Tasks are enabled by events that may be received from other tasks, event channels or at regular intervals from a timer. The scheduler triggers the execution of tasks based on the scheduling policy. After execution, the task may also initiate other tasks by generating a `start` event. We assume that the task generates events at the end of the execution – this assumption is valid in several component middleware (i.e. CCM [25] or J2EE [31]) where transactions are committed at the end of the computation.

4.3.3. The Event Channel Event propagations between tasks follow a non-blocking broadcast semantics – events which are not received are lost. Therefore, event passing has to be synchronized between the publisher and consumer tasks. To alleviate these restrictions communication between tasks are coordinated through event channels.

Event channels provide the necessary mechanism to allow asynchronous communication – the publisher does not have to block until the consumer is ready to receive the event. Published events are stored in the event channel until the consumer is ready to receive them.

The state denoted with the **C** letter are *committed* states. This expresses time constraints, the local clock of a timed automaton model cannot be increased in a *committed* state, in other words an outgoing transition has to be taken from these states, otherwise the system

will deadlock.

Event channels are generically represented by the timed automaton shown in Figure 2. Note that this model takes into account possible communication delay as represented by the channel dependent maximum delay factor δ_c.

4.3.4. The Scheduler To express the mapping of execution tasks to platform processors we introduce the scheduler modeling construct shown in Figure 2. The scheduler selects enabled tasks for execution and triggers preemptions according to the scheduling policy.

The scheduler initially starts in the `idle` state. It will move to the `select` state if any tasks become eligible for execution. The selection is made instantaneously from the of enabled tasks' queue and the selected task will receive the `run` event which triggers its execution. The time required for selection and context switching can be bounded in a real-time implementation and added to the measured `WCET` times of the tasks. The scheduler moves to the wait (initial) state if no task is ready for execution.

The scheduling policy is encoded in three functions: (1) $\mathsf{Add}(w, i)$, which increases the current priority level when task$_i$ becomes ready, (2) $\mathsf{Sub}(w, i)$, which decreases the current priority level when task$_i$ becomes ready, and (3) $\mathsf{Enable}(w, i)$, which evaluates to `true` if the ith task is eligible for execution. For example, in the case where priority is directly proportional to the component index, $\mathsf{Add}(w, i) = w + 2^{i-1}$, $\mathsf{Sub}(w, i) = w - 2^{i-1}$, and $\mathsf{Enable}(w, i) = 2^{i-1} \leq w < 2^i$. Other scheduling schemes can be established by defining appropriate formulas for the three functions outlined above.

5. Case Study

In this section first we briefly introduce the architecture of an avionics application then we show how we applied model-based methods to verify the preemptive scheduling on a distributed, multi-processor platform.

5.1. The Boeing Bold Stroke Architecture

The Boeing Bold Stroke architecture is a component-based, *publish/subscribe* DRE system platform built atop *The ACE ORB* [29]. Bold Stroke uses a Boeing-specific real-time component middleware called *PRISM* [27]. Following the CCM specification, PRISM defines typed *ports*, which are named interfaces and connection points components use to collaborate with each other. Operation invocations follow the traditional synchronous blocking *call/return* semantics, where one component's receptacle is used to invoke an operation on another component's facet. PRISM's event propagation mechanism follows a non-blocking asynchronous *publish/subscribe* semantics supported by real-time event channels connected via event sources/sinks.

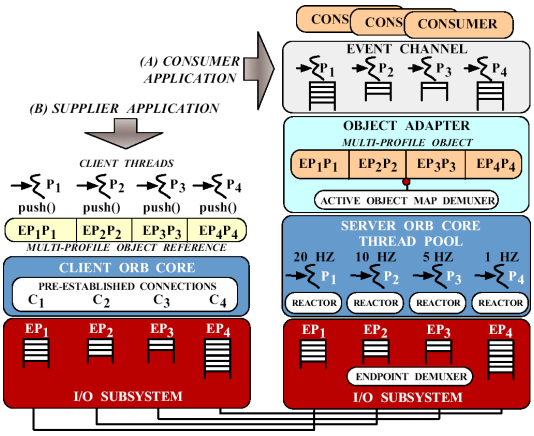

Figure 3. Boeing Bold Stroke Execution Platform

Figure 3 shows the runtime architecture of the Bold Stroke execution platform, which consists of three primary layers: (1) the *ORB layer*, which performs (de)marshalling, connection management, data transfer, event/request demultiplexing, error handling, concurrency, and synchronization, (2) the *real-time event channel* layer, which schedules and dispatches events in accordance with their deadlines and other real-time properties, and the *application component layer*, which contain actions that are the smallest units of end-to-end processing that Bold Stroke application developers can manipulate. The concurrency architecture implements the *Half-Sync/Half-Async* pattern [30], where a fixed pool of threads is generated in the server at the initialization phase to process incoming requests.

Bold Stroke actions are largely event-driven and use priority-based scheduling, where actions that have the same priorities are scheduled non-preemptively in a *priority band* (also referred to as *rate group*) based on the queueing policy (FIFO, MUF etc.) This queueing policy is modeled by introducing *subpriorities* for the tasks. Preemptive scheduling is used between priority bands which usually correspond to individual threads. Periodic real-time processing of frames is driven by unsynchronized software timers that can drift apart from each other, so component interactions are unrestricted

and unsynchronized. This approach is intentional [9] and designed to increase flexibility and performance, though it has the side-effect of impeding analyzability.

We have shown in our previous work [24,23] that the smallest unit of scheduling is an event-initiated action together with all the remote calls it can invoke, since they inherit the QoS properties of the calling thread. Every action has a measured *worst-case execution time* (`WCET`)[1] in the given scenario in which it is used.

All processing inside a priority band must finish within the fixed execution period of the timer assigned to the band. This periodicity divides processing into *frames*, which are defined by the rate/period of the timer. A priority band failing to complete outputs prior to the start of the next frame is said to incur a *frame overrun* condition, where the band did not meet its completion deadline (*i.e.*, frame completion time).

5.2. Distributed Bold Stroke Application

In this section we show how we applied model-based verification to a case study of a representative DRE system from the domain of avionics mission computing. Figure 4 shows the component-based architecture of the system, which is built on the Boeing Bold Stroke real-time middleware described in Section 5.1.

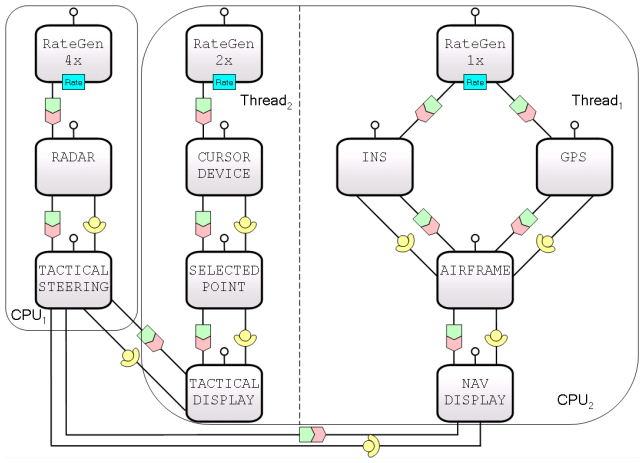

Figure 4. Bold Stroke application deployed on a preemptive multiprocessor platform

1 `WCET`s are computed by measuring the times corresponding to executing the tasks numerous times in a loop. `WCET` times do not include the time spent waiting to be scheduled and are assumed independent of the scheduling policy.

The application is deployed on a preemptive multiprocessor platform. The `RateGen 4x`, `RADAR` and `TACTICAL_STEERING` components are deployed on the same application server ($\texttt{CPU}_1$) and are scheduled non-preemptively. The other application server ($\texttt{CPU}_2$) is a multi-threaded server, where priorities are assigned to the threads. The real-time middleware maps CORBA priorities to operating system-level priorities and the OS scheduler manages preemptions.

The `RateGen 2x`, `CURSOR_DEVICE`, `SELECTED_POINT` and `TACTICAL_DISPLAY` components are executed within the context of the higher priority $\texttt{Thread}_2$, the `RateGen`, `INS`, `GPS`, `AIRFRAME` and `NAV_DISPLAY` components are executed within the context of the lower priority $\texttt{Thread}_1$.

Event propagations are denoted with small arrows in Figure 4 which represent the binding between event sources and sinks. Facets are represented by circles which connect to the receptacles. We observe the following key challenges on the Bold Stroke example:

- *Event flow, buffering:* Event propagations require buffering of the events (i.e. for the `AIRFRAME` component) and concurrency management between event channels which are publishing to the same component (i.e. between the remote and local event channels which publish events to `NAV_DISPLAY`).
- *Delays:* Communication between application servers incur delays in the message propagation.
- *Preemption:* We have to manage concurrency between threads inside the application server (i.e. $\texttt{Thread}_1$ is periodically preempted by $\texttt{Thread}_2$).
- *Clock drifts:* Timer components are not synchronized and can drift from each other arbitrarily.
- *Composition:* The problems above can be summarized as composition challenges. *The schedulability of individual threads do not guarantee the overall schedulability of the system.*

5.3. Preemption Model Using Discretized Time

As discussed in Section **??** we provide a conservative approximation of the actual scheduling policy by discretizing the time scale during preemptions. When implementing this discretization, we had a choice whether to (1) implement *blocking* preemptions, when the high priority thread has to block until the next discrete time

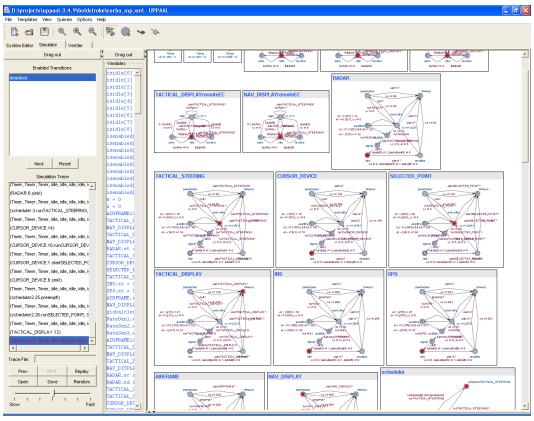

Figure 5. Detected Deadlock with Counter Example

step arrives, or (2) *non-blocking* preemptions when preemptions take place instantaneously but all information since the last time step is lost in the preempted thread. The first case implies longer execution time (than the parameter `WCET` time) in the high-priority thread (most often the `Timer`), the second case implies longer execution time in the low priority thread.

We chose to implement non-blocking preemptions for three major reasons; (1) higher precision is desirable on higher priority threads, (2) a blocking thread will propagate its precision loss to every consumer, whereas on non-blocking preemptions we only lose precision on one consumer, (3) by blocking `Timers` we would unintentionally synchronize them. The precision can be set independently for every Uppaal `Task` model using the `granularity` parameter.

5.4. Compositional Analysis Using Uppaal

To check whether a property holds we use the Uppaal model checker tool [26]. In addition to simulation, Uppaal provides built in support for manual and automatic simulation. To improve efficiency, the model checking algorithms in Uppaal are based on clock constraints equivalence rather than state equivalence. Systems in Uppaal are modeled as a slightly modified variant of timed automata and the specification is expressed in a restricted version of the *timed computational tree logic* (TCTL) [5], which is temporal logic that can formalize statements about system models. The Uppaal semantic domain combines timed automata with dataflow semantics that can be used to express interactions between the automata.

Figure 6 shows how we modeled the system in the Uppaal model checker tool. The application consits of 12 components. The `RateGen 1x`, `RateGen 2x` and `RateGen 4x` components are simple rate generators which publish events at a predefined rate. We model them using `Timers` in the DRE Semantic Domain. The other 9 components are modeled using `Tasks` in the DRE Semantic Domain.

To satisfy real-time constraints the PRISM component middleware requires dedicated threads for each real-time event channel – to avoid unnecessary thread spawn delays. In the DRE Semantic Domain, however, we can abstract out some of these threads as discussed in [24, 23], to reduce the number of event channels and thus the state space. We have to model event channels explicitly (1) when we have to buffer events or (2) on remote event channels which have measureable delays. We introduce the `AIRFRAMElocalEC`, `NAV_DISPLAYlocalEC`, `TACTICAL_DISPLAYlocalEC` channels to model buffering during local event propagations and the `TACTICAL_DISPLAYremoteEC`, `NAV_DISPLAYremoteEC` channels to express delays between the application servers denoted by CPU_1 and CPU_2 on Figure 4.

The scheduling policies are represented by `Schedulers` in the DRE Semantic Domain. Since the Bold Stroke application is deployed on a two-processor architecture we define two schedulers as shown on Figure 6. The schedulers get more and more complex according to the scheduling policies. The design models (such as the model on Figure 4) can be used to generate the scheduling policies as shown in [24, 23].

Now that we have defined the elements in the DRE Semantic Domain we show how their *composition* can aid DRE system developers in the design and build process. We start modeling the Bold Stroke application shown in Figure 4 by composing same-priority components and local event channels corresponding to a thread. Priorities follow dependencies and are therefore fixed. We assign deadlines according to Theorem 1. Table 1 shows the parameter values.

In the next step we compose the components to reflect the architecture of the implementation. This step follows the imperative approach, while the task properties are declarative. We give a granularity value of 1 to every task. Most tasks which never get preempted do not need higher precision. Then we bind tasks with remote event channels.

When checking the Bold Stroke application shown in Figure 4 we were unable to show the schedulability with rough precision (`granularity = 1` for every task). This does not imply an unschedulable sys-

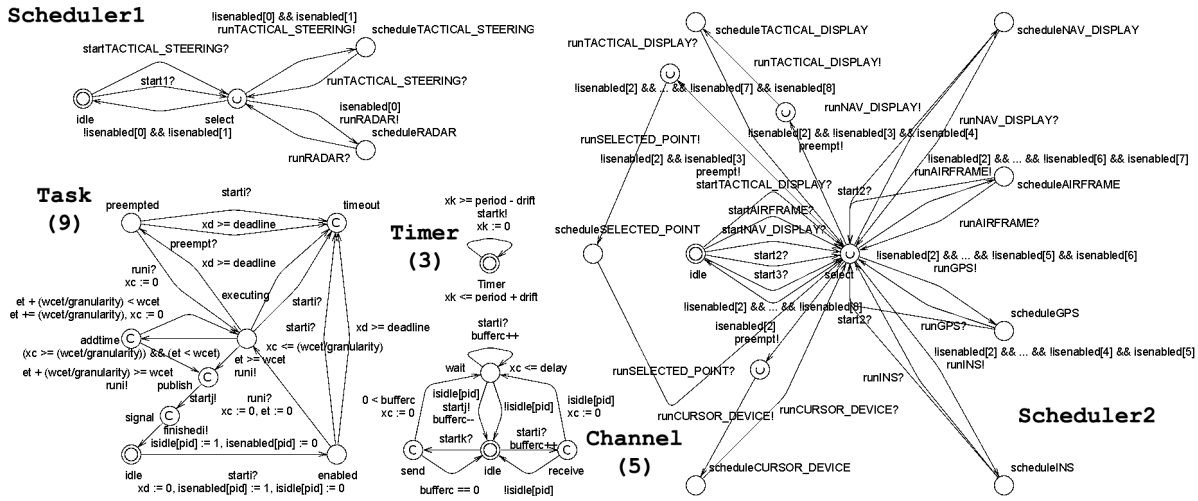

Figure 6. Uppaal Timed Automata Models

Task	P	S	WCET	D	G
RADAR	HIGH	HIGH	12	84	1
TACT_ST...	HIGH	MEDIUM	16	88	1
CURS_D...	HIGH	HIGH	18	155	1
SEL_P...	HIGH	MEDIUM	24	161	1
TACT_D...	HIGH	LOW	21	158	1
INS	LOW	HIGH	32	872	32
GPS	LOW	HIGH	29	869	29
AIRFRAME	LOW	MEDIUM	80	920	80
NAV_D...	LOW	LOW	19	859	19

Table 1. Parameter Values for the Application Shown in Figure 4

tem since we are approximating the real behavior. In the next step we have increased the granularity parameters to 1 millisecond precision in the preempted tasks as shown in Table 1. This analysis has shown that the system is schedulable with the parameters given in 1. In the next series of tests we have checked whether the system operates with finite buffer sizes with the following TCTL formula: `A[] (`*`Channel`*`.bufferc < `*`Channel`*`.lambdac)`. UPPAAL produces a counter-example for invalid properties as shown on Figure 5, which helps identifying the source of undesired behavior. Finally, we checked that eventually every task will execute with: `E<>` *`Task`*`.executing`.

6. Related Work

Several tools and methods have been developed for the design, optimization, analysis and verification of scheduling of various distributed real-time embedded (DRE) systems. To deal with the space constraints, we just mention key results of the research on Bold Stroke scheduling and verification, which are the VEST, AIRES, CADENA and TIME WEAVER - TIMEWIZ® tools.

The Virginia Embedded Systems Toolkit (VEST) [17] is a framework designed for the reliable and configurable composition and analysis of component-based embedded systems from COTS libraries. VEST applies key checks and analysis but does not support formal proof of correctness.

The Automatic Integration of Reusable Embedded Systems (AIRES) tool extracts system-level dependency information from the application models, including event- and invocation-dependencies, and constructs port- and component-level dependency graphs. It performs real-time analysis [12] using Rate Monotonic Analysis techniques.

The CADENA [13] framework is an integrated environment for building and analyzing CORBA Component Model (CCM) based systems. Its main functionalities include CCM code generation in Java, dependency analysis and model-checking. The emphasis of verification in Cadena is on software logical properties. The generated transition system does not represent time explicitly and requires the modeling of logical time that does not allow quantitative reasoning.

TIME WEAVER (GEODESIC) [8] is a component-based framework that supports the reusability of com-

ponents across systems with different para-functional requirements. It supports code generation as well as automated analysis. It performs model-based Rate-Monotonic Analysis by real-time model checker tools such as TIMEWIZ®.

7. Concluding remarks

This paper presents the DRE SEMANTIC DOMAIN used in the next generation OPEN-SOURCE DREAM [16] model-based verification and analysis framework. We have shown how to analyze and verify the *preemptive scheduling* and *composition* of real-time embedded systems on unsynchronized distributed platforms and how model-based verification provides a way to bridge the gap between *declarative specification* and *imperative implementation*. Our results show that this approach captures *delays* and *drifts* which are common in *unsynchronized reactive real-time systems* and demonstrates that timed automata can represent component interactions and asynchronous event passing allowing the verification of quantitative dense time properties. The verification is automatic, exhaustive, and capable of producing counter-examples that helps pinpoint sources of undesired behavior. The proposed method is demonstrated on the Boeing Bold Stroke avionics mission computing platform.

Key challenges for the upcoming OPEN-SOURCE DREAM framework include optimizing *multiple QoS properties*, thereby reducing nondeterminism in design-time allowing better analyzability. Integrating real-time middleware and multiple analysis tools into the OPEN-SOURCE DREAM framework forms a key part of our future work. The Uppaal models used for the case study in Section 5 are available for download at [16].

References

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, 2003.

[2] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. *Formal Methods for the Design of Real-Time Systems, LNCS 3185*, pages 237–267, Sep 2004.

[3] C. Brooks, A. Cataldo, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, and H. Z. (eds.). Hyvisual: A hybrid system visual modeler. Technical report, University of California, Berkeley, UCB ERL Technical Memorandum UCB/ERL M04/18, 2004.

[4] J. Buck, S. Ha, E. A. Lee, and D. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation, special issue on Simulation Software Development*, 4:152–182, April 1994.

[5] E. Clarke and E. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. *Logic of Programs, Lecture Notes in Computer Science*, 131:52–71, 1981.

[6] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the futurebus+ cache coherence protocol. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, pages 15–30. North-Holland, 1993.

[7] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219. Springer-Verlag New York, Inc., 1996.

[8] D. de Niz and R. Rajkumar. Time Weaver: A Software-Through-Models Framework for Real-Time Systems. In *Proceedings of LCTES*, 2003.

[9] B. S. Doerr and D. C. Sharp. Freeing Product Line Architectures from Execution Dependencies. In *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[10] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[11] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions, 2003.

[12] Z. Gu, S. Wang, S. Kodase, and K. G. Shin. An End-to-End Tool Chain for Multi-View Modeling and Analysis of Avionics Mission Computing Software. In *Proceedings of Real-Time Systems Symposium*, 2003.

[13] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of International Conference on Software Engineering*, 2003.

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Lecture Notes in Computer Science*, 2211:166+, 2001.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM Press.

[16] http://dre.sourceforge.net. Distributed Real-time Embedded Analysis Method. 2005.

[17] J.A. Stankovic and R. Zhu and R. Poornalingham and C. Lu and Z. Yu and M. Humphrey and B. Ellis. VEST: An Aspect-based Composition Tool for Real-time Systems. In *Proceedings of the IEEE Real-time Applications Symposium*, 2003.

[18] R. John Hatcliff, Matthew B. Dwyer. Bogor: An Extensible Framework for Domain-Specific Model Checking. Technical report, Kansas State University, December 2004.

[19] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-Integrated Development of Embedded Software. *IEEE*, 91:145–164, Jan. 2003.

[20] P. Krčál and W. Yi. Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata. In K. Jensen and A. Podelski, editors, *Proc. of TACAS'04, Barcelona, Spain.*, volume 2988 of *Lecture Notes in Computer Science*, pages 236–250. Springer–Verlag, 2004.

[21] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[22] E. A. Lee. Concurrent Models of Computation for Embedded Software. Technical report, University of California, Berkeley, UCB ERL Technical Memorandum UCB/ERL M05/2, 2004.

[23] G. Madl, S. Abdelwahed, and G. Karsai. Automatic Verification of Component-Based Real-Time CORBA Applications. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 231–240, December 2004.

[24] G. Madl, S. Abdelwahed, and D. C. Schmidt. Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking (invited paper, submitted). *The International Journal of Time-Critical Computing*, 2005.

[25] Object Management Group. CORBA Component Model. 2002.

[26] P. Pettersson and K. G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, feb 2000.

[27] W. Roll. Towards Model-Based and CCM-Based Applications for Real-Time Systems. In *ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)*, pages 75–82. IEEE Computer Society, 2003.

[28] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. In M. D. Cin, C. Meadows, and W. H. Sanders, editors, *Dependable Computing for Critical Applications—6*, volume 11, pages 203–222, Garmisch-Partenkirchen, Germany, 1997. IEEE Computer Society.

[29] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar. A High-Performance Endsystem Architecture for Real-Time CORBA. *IEEE Communications Magazine*, 14(2), 1997.

[30] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2.* Wiley & Sons, New York, 2000.

[31] Sun Microsystems. Java 2 Platform, Enterprise Edition (J2EE) v1.4. 2003.

[32] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, pages 110–112, Apr. 1997.

[33] J. Sztipanovits, G. Karsai, and H. Franke. Model-Integrated Program Synthesis Environment. In *Proceedings of the IEEE Symposium on Engineering of Computer Based Systems*, pages 348–355, March 2000.