Verifying Distributed Real-time Properties of Embedded Systems via Graph Transformations and Model Checking*

Gabor Madl[1,2] Sherif Abdelwahed[1] Douglas C. Schmidt[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

Component middleware provides dependable and efficient platforms that support key functional, and quality of service (QoS) needs of distributed real-time embedded (DRE) systems. Component middleware, however, also introduces challenges for DRE system developers, such as evaluating the predictability of DRE system behavior, and choosing the right design alternatives before committing to a specific platform or platform configuration. Model-based technologies help address these issues by enabling design-time analysis, and providing the means to automate the development, deployment, configuration, and integration of component-based DRE systems. To this end, this paper applies model checking techniques to DRE design models using model transformations to verify key QoS properties of component-based DRE systems developed using Real-time CORBA. We introduce a formal semantic domain for a general class of DRE systems that enables the verification of distributed non-preemptive real-time scheduling. Our results show that model-based techniques enable design-time analysis of timed properties and can be applied to effectively predict, simulate, and verify the event-driven behavior of component-based DRE systems.

Keywords: Schedulability analysis, model checking, component middleware, distributed real-time and embedded systems.

1 Introduction

Distributed real-time embedded (DRE) systems, including command and control systems, and avionics mission computing, increasingly run in open environments, such as network-centric systems of systems. These open DRE systems operate in less predictable conditions than previous generations of real-time and embedded systems (such as micro-controllers and PLCs) that are specialized for specific application domains (such as heavy industry, traffic control, or medical applications). 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, thereby off-loading many tedious and error-prone aspects of the software lifecycle from developers of DRE systems.

A promising software infrastructure technology for DRE systems is *component middleware*, which defines platform capabilities, and tools for specifying, implementing, deploying, and configuring *components* [1], and publish/subscribe services [2] that exchange messages between components. Components, in

*This research was supported by the NSF Grants CCR-0225610 and ACI-0204028

turn, are units of implementation, reuse, and composition that expose named interfaces called *ports*, which are connection points that components use to collaborate with each other. Component middleware helps simplify the development and validation of DRE systems by providing reusable services, and optimizations that support their functional, and quality of service (QoS) needs more effectively than conventional *ad hoc* software implementations.

Despite recent advances in component middleware, however, there remain significant challenges that make it hard to develop large-scale DRE systems for domains that require simultaneous support for multiple QoS properties, such as low latency, low energy consumption, bounded jitter, high reliability, high throughput, and scalability. Examples of such systems include total ship computing environments, and supervisory control and data acquisition (SCADA) systems that manage regional power grids. Key unresolved challenges include the lack of tools for effectively configuring, integrating, and verifying DRE systems built using components [3].

To address these challenges, it is useful to analyze system behavior early in the lifecycle, thereby enabling developers to select suitable design alternatives before committing to specific platforms or implementations. In particular, making these decisions in the design phase helps minimize mistakes that would otherwise be revealed in the testing and integration phases, when they are much more expensive to fix. Design-time analysis requires a means of expressing component behavior with respect to their QoS properties, defining the semantics for component interactions, and composing components to form subsystems.

It has historically been hard to analyze DRE systems developed using component middleware due to the many accidental complexities associated with component middleware, and component-based applications written using third-generation programming languages, such as C++ and Java. Key disadvantages of analysis methods based on source code are that they require a working implementation, and must also overcome accidental complexities with popular third-generation programming languages features, such as pointers and dynamic memory allocation. When analyzing DRE systems, we focus on their structure, behavior, environment, and the properties they must satisfy. The structure of the system is usually hard to reconstruct from source code since third-generation languages give an informal, and often ambiguous definition of behavior. Moreover, they do not capture the desired operational properties of the system (such as liveness, graceful degradation, and dependability) because they describe the *behavior* of the system at too low a level of abstraction.

To reduce this complexity – and facilitate more powerful and extensive analysis – it is essential to raise the abstraction level from low-level implementation details to higher-level models of DRE system structure and behavior. Identifying and validating abstractions that support this evolution effectively is the key to successful *model-based verification* [4,5]. Formal methods are mathematically-based languages, techniques, and tools for specifying and verifying complex systems. By *formalizing the specifications* and the behavior of these systems, we can increase their analyzability significantly. This paper introduces a new model-based approach for verifying non-preemptive scheduling of DRE systems, as follows:

- We describe a reusable framework for model-based verification called the *Distributed Real-time Embedded Analysis Method* (Dream). Dream is a platform-independent, generic and extendable framework for the simulation and verification DRE systems. Dream builds on the timed automata formalism and uses model checking tools for schedulability analysis. This analysis method can be integrated flexibly with refinement-based system development, as well as with platform-based design automation.

- We present the DRE Semantic Domain, which is formalizes the key timed properties of DRE systems based on timed automata. We use the DRE Semantic Domain to verify distributed non-preemptive real-time scheduling of avionics applications built upon the Boeing Bold Stroke platform [6,7], which is component middleware developed using Real-time CORBA [8] that has been applied in a variety of mission-critical avionics computing systems. This specification technique enables the use of a wide variety of modeling formalisms supported by analysis tools, such as Uppaal [9,10], Kronos [11], and the IF [12] toolset.

- We present an analysis method that uses Dream to capture the reactive behavior of event-driven DRE systems, and verify critical timed properties. We show that Dream can verify key QoS properties for DRE systems, such as end-to-end deadlines, graceful degradation, or dependability. Our techniques can be used to analyze component-based DRE systems based upon (1) relatively tightly coupled time-triggered periodic processing, (2) more loosely coupled aperiodic event-driven publish/subscribe communication patterns, and (3) systems with a both time-triggered and event-driven dynamics.

This paper uses the Dream framework to illustrate and validate the concepts of model-based verification in the context of the Boeing Bold Stroke platform. In particular, we consider the problem of deciding the schedulability of a given set of Bold Stroke tasks with event- *and* time-driven interactions. We represent the task model and scheduling policy via the DRE Semantic Domain model of computation. The timed automata formulation of the problem translates the schedulability problem into a reachability problem in which the set of tasks are schedulable if none of the corresponding timed automata can reach a state that was predefined to express missed deadlines. If this analysis completes successfully it implies that all tasks complete before their respective deadlines.

The remainder of this paper is organized as follows. Section 2 describes the Boeing Bold Stroke execution framework; Section 3 formalizes the DRE Semantic Domain and summarizes the underlying research problems; Section 4 explains our model-based Dream verification method; Section 5 uses Bold Stroke as a detailed case study of model-based verification; Section 6 compares our research on model-based verification with related work, and Section 7 presents concluding remarks.

2 Boeing Bold Stroke Execution Platform

The Boeing Bold Stroke architecture is an event-driven component-based DRE system platform built atop (1) *The ACE ORB (TAO)* [13], which implements key Real-time CORBA [8] features (such as threadpools, lanes, and client-propagated and server-declared threading policies), and (2) TAO's Real-time Event Service [2], which implements the publish/subscribe pattern [14], and schedules and dispatches events via a federation of real-time event channels (any event channel mentioned in the rest of the paper refers to the real-time event channel). Bold Stroke uses a Boeing-specific component model called *PRISM* [7], which implements a variant of the CORBA Component Model (CCM) [1] atop TAO. Following the CCM specification, PRISM defines the following types of *ports*, which are named interfaces, and connection points components use to collaborate with each other:

- **Facets**, which define named interfaces that process method invocations from other components.

- **Receptacles**, which provide named connection points to facets provided by other components.

- **Event sources and event sinks**, which indicate a willingness to exchange event messages with one or more components via event channels.

PRISM operation invocations provides blocking synchronous *call/return* semantics, where one component's receptacle is used to invoke an operation on another component's facet. Conversely, PRISM's event propagation mechanism provides non-blocking asynchronous *publish/subscribe* semantics supported by real-time event channels connected via event sources/sinks. When a publisher pushes an event to an event channel all of its subscribed components are notified.

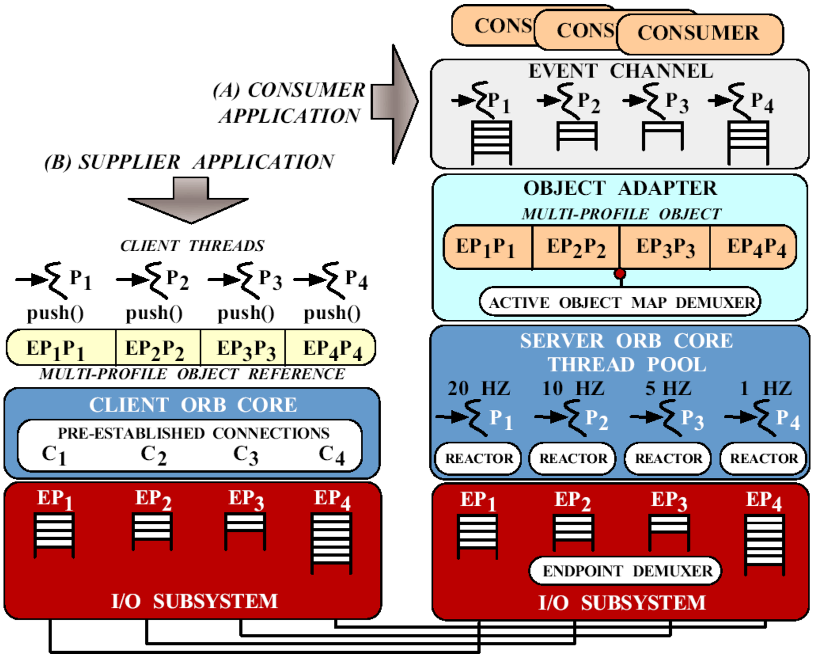

Figure 1: The Boeing Bold Stroke Execution Platform

Although the CCM specification allows the dynamic creation and connection of components, PRISM follows common patterns in safety/mission-critical systems, and enforces a static component allocation and configuration policy by creating and connecting components only during system initialization. Dynamical components in PRISM can reconfigure themselves by changing their behavior based on system mode settings, such as takeoff mode, landing mode, and threat-evasion mode.

Figure 1 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 (3) the *application component layer*, which contain actions that are the smallest units of end-to-end processing that Bold Stroke application developers can manipulate.

Bold Stroke actions are largely event-driven, rather than strictly time-triggered. In particular, periodic real-time processing of frames is driven by unsynchronized software timers that may drift apart from each other, so component interactions are unrestricted and unsynchronized. This approach is intentional [15] and designed to increase flexibility and performance, though it has the side-effect of impeding analyzability and strict predictability.

As a result of Bold Stroke's event-driven architecture, dependencies between actions can significantly influence the schedulability of avionics mission computing systems built atop it. Bold Stroke applications 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 their sub-priorities. In this setting, preemptive scheduling is used between priority bands, whereas non-preemptive scheduling is used within a particular band.

A priority band is implemented by three types of threads: (1) the *dispatcher (worker) threads*, which reside in real-time event channels and execute all actions initiated by event propagations, (2) the *interval timeout thread*, which pushes timeout events at predefined intervals, and (3) *ORB threads*, which continually process request inputs from the ORB core executing actions initiated by operation invocations. This concurrency architecture implements the *Half-Sync/Half-Async* pattern [16], where a fixed pool of threads is generated in the server at the initialization phase to process incoming requests. This pattern ensures that Bold Stroke applications incur low latency and jitter for end-to-end actions [17].

An action has an assigned priority and sub-priority *(importance)* value for every real-time event channel to which it is subscribed. If two actions have the same sub-priority they will be ordered or scheduled non-deterministically according to the configuration. Every action has a *worst-case execution time* (`WCET`)[1] in the given scenario in which it is used. Actions can be initiated by two ways: operation invocations and event propagations. The Bold Stroke scheduling strategy is also configurable – by default its actions are scheduled in accordance with *Rate Monotonic Analysis* [18].

Facet-initiated actions invoked by a remote operation call inherit the QoS execution semantics from the invoking component and do not interact with TAO's runtime scheduler, which resides inside the real-time event channels. We therefore do not distinguish these actions from the invoking action in the scheduling perspective. The smallest unit of scheduling is an event-initiated action together with all the remote operation calls it can invoke. Since facet-initiated actions can also call other actions using remote operation calls, the complete call chain is an acyclic graph, with the event-initiated action as the root element. We call this smallest unit of scheduling an *invocation unit.*

An executing action may initiate actions on other priority bands, which are known as *cross-rate actions.* 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. For example, a 20 Hz timer will have a 50 ms frame and the overall execution time of the tasks in the timer's rate group must be smaller than 50 ms to fit within the frame. A priority band failing to complete outputs prior to the start of the next frame incurs a so-called *frame overrun* condition, where the band did not meet its completion deadline, *i.e.*, the frame completion time.

3 The DRE Semantic Domain

This subsection formalizes a computational model that can express the event-driven nature of DRE systems with characteristics similar to those described in Section 2. We define a model on a distributed platform with non-preemptive executions. The proposed model of computation is timed automata. We refer to this semantic domain as the DRE Semantic Domain. We chose timed automata as the underlying model of computation for our analysis since it has well-defined formal semantics [19], is supported by several automated model checking tools, and is expressive enough to capture the dynamics of a wide class of DRE systems.

The DRE Semantic Domain can be used to model basic components in DRE system such as timers, dynamic computation tasks, event channels, and schedulers. DRE system models can be built by the composition of these components. To facilitate formal developments – and without loss of generality – we assume that communication through shared variables or common coupling are not considered in the model. We also assume that 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.

This paper considers the problem of deciding the schedulability of a given set of tasks with event- *and* time-driven interactions on a distributed non-preemptive platform. In the context of DRE systems in which each task is associated with a deadline, a system is *schedulable* if all tasks finish their execution before their respective deadlines. In our work the deadline is measured from the time when the task is enabled, *i.e.*, the time when it receives a `release` event. We use a timed automata formulation of the problem that translates the schedulability problem into a reachability problem in which the set of tasks

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

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.

3.1 Timed Automata

This section reviews timed automata as the underlying semantic domain for the proposed analysis. A timed automaton is a state machine equipped with a finite set of real-valued clock variables, called *clocks* for short. We use $\mathcal{AC}(V)$ to denote the set of atomic constraints over the set of variables V where c_i $\mathtt{op}$ n_a or $c_i - c_j$ $\mathtt{op}$ n_b where $c_i, c_j \in V, \mathtt{op} \in \{<, \leq, >, \geq\}, n_a, n_b \in \mathbb{N}$. We use $\mathcal{RC}(V)$ to denote the restricted subset of $\mathcal{AC}(V)$ where all atomic constraints are expressed as x_i $\mathtt{op}$ $n_i, \mathtt{op} \in \{<, \leq\}$. The set of atomic constraints $\mathcal{AC}(V)$ represents the *clock constraints* or *guards.*

Transitions in a timed automaton can have guards and reset operations on clock and data variables. Transitions are enabled if the corresponding guard evaluates to `true`. An enabled transition can execute instantaneously while resetting certain variables to new values according to the underlying reset assignments. States in timed automata can be associated with an invariant that determines the validity of clock assignment in the state. A system can be in a given state only if the underlying invariant is `true`.

Definition 1 *A timed automaton is a 5-tuple $\mathcal{TA} = (C, L, l_0, E, Inv)$ consisting of the following components:*

- *a finite set of clock variables C. Each clock variable $c \in C$ is evaluated over* $(\mathbb{R}^+ \cup 0)$*;*
- *a finite non-empty set L of vertices called* locations*;*
- *the initial location $l_0 \in L$;*
- *a finite set of edges $E \subseteq L \times \mathcal{AC}(C) \times 2^C \times L$ called transitions, where $e = (l, \gamma, \alpha, l')$ represents an edge from location l to location l' with guard γ and α which is an assignment of clocks to the value* zero *(possible clock reset);*
- *a labeling function $Inv : L \rightarrow \mathcal{RC}(C)$ that assigns a restricted atomic constraint to each location. This constraint is the invariant of the location.* □

Definition 2 *A state of the timed automaton is defined as a pair (l_i, c_i) where $l_i \in L$ and c_i is a valuation of the clock $c_i \in C$. We denote the set of states as S.* □

A more detailed description of the timed automata model can be found in [19]. The Uppaal model checking tool uses an extended version of the above timed automata formulation. The Uppaal extensions to the timed automata model of computation are described in [9].

3.2 Timed Languages

We extend the definition of timed automata with two sets of states: *regular* and *error* states. A finite (infinite) run ρ is accepting if ρ does not contain any of the *error* states (and it contains infinitely many regular states). With every finite (infinite) run we associate a timed word σ: with each transition we associate a *timestamp* t_i, which is the value of the global clock in C. The run ρ therefore accepts the timed word $(a_0, t_0), (a_1, t_1), \ldots, (a_n, t_n)$. The *timed language* $\mathcal{L}(\mathcal{TA})$ accepted by the timed automata $\mathcal{TA}$ is the set of timed words which correspond to accepting runs.

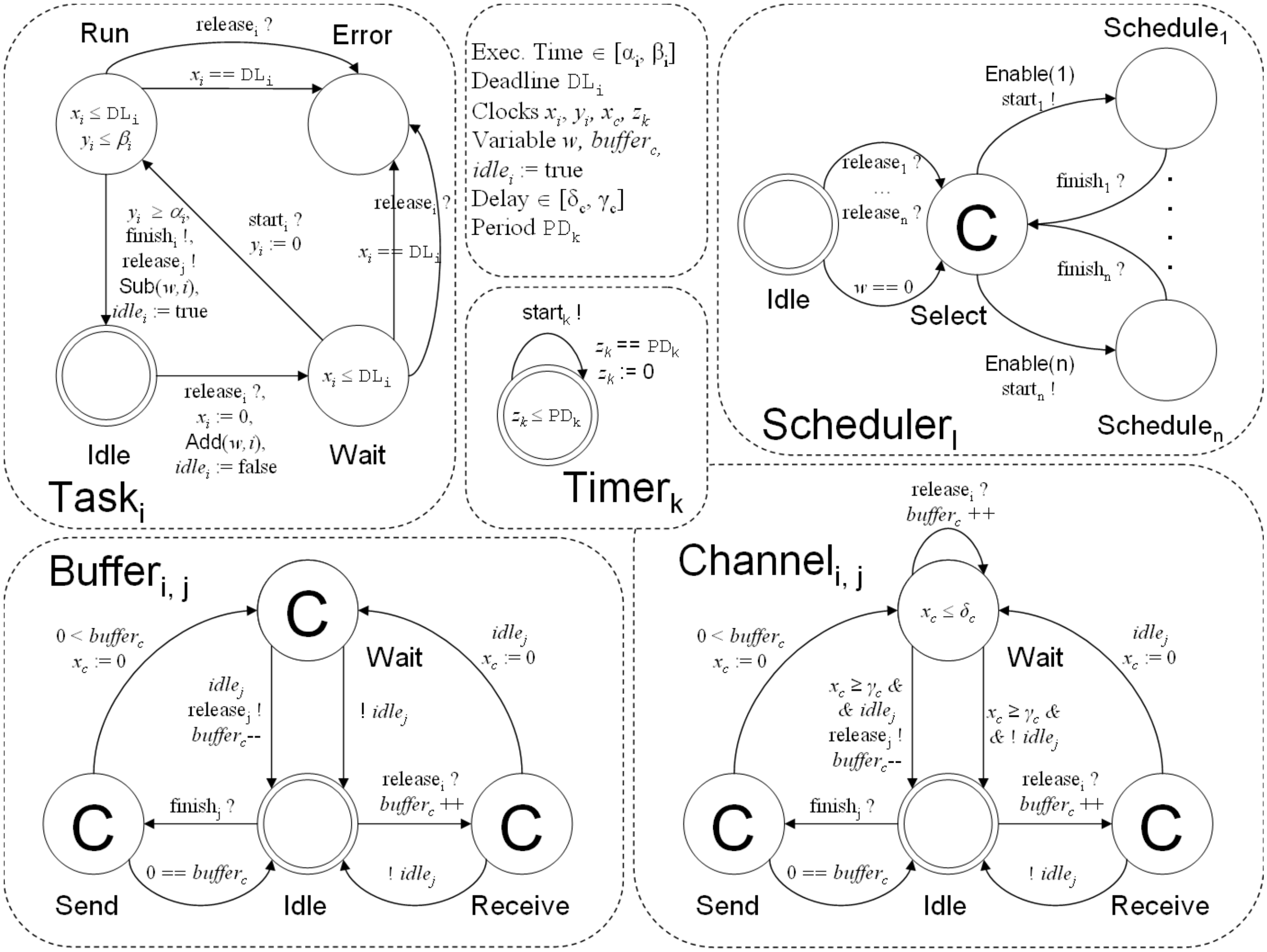

Figure 2: Generic Model of the DRE Semantic Domain

3.3 Timers

A timer in the Dream semantic domain is a simple periodic event generator that releases task initiation events at a specified rate. Timers may represent sensors sampled at a predefined rate and can drift arbitrarily from each other. We assume, however, that time drifts in timers are bounded. Timers can be represented by a generic timed automaton model shown in Figure 2.

3.4 Tasks

Tasks are the main components in DRE systems that carry out computations and interact with the physical environment. 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, a task may also initiate other tasks by generating a `release` event. We assume that the task generates events at the end of the execution. Generating an event at the end of the execution is optional.

3.5 Event Channels

Event propagations between tasks follow the timed automata non-blocking broadcast semantics, *i.e.*, events that are not received (and therefore lost) are not regenerated since received events are not acknowledged. To provide a reliable event service, therefore, event passing must be synchronized between the publisher and consumer tasks. To alleviate these restrictions, communication between tasks are coordinated through event channels that provide the mechanisms to allow reliable asynchronous communication, where publishers do not block until the consumer receives the event. Instead, published events are queued in the event channel until the consumer is ready to receive them.

Event channels are generically represented by the timed automaton shown in Figure 2. This model takes into account possible communication delays represented by the channel dependent maximum delay factor δ_c and minimum delay factor γ_c. The locations denoted with the `C` letter correspond to committed locations. In the UPPAAL timed automata model, a location is committed if an outgoing transition must be taken as soon as the system enters this location.

An event channel receives events from a timer or task and propagates it to a task with some delay $\delta \in [\gamma_c, \delta_c]$. If the task is in an execution cycle (not in the `idle` location) it buffers the events. When the task receives a `release` event it either propagates it or buffers it according to the state of the subscribed task. When a subscribed task finishes the execution cycle the event channel moves to the `wait` location if the buffer is not empty. After the delay δ has passed the event channel generates an event for the subscribed task if the subscribed task is in the `idle` location. When an event is received in the `wait` location the event channel buffers it.

The `Buffer` modeling construct shown in Figure 2 is a special version of the event channel that ensures every event will be delivered before the scheduler is invoked. This construct is used to model event channels within a thread as explained in Section 5.

3.6 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 according to the scheduling policy. The scheduler initially starts in the `idle` location and will move to the `select` location if any tasks become eligible for execution. The selection is made instantaneously from the enabled tasks' queue and the selected task will receive the `start` 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 `idle` location 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.

3.7 Event propagation

This section explains how events propagate in the DRE SEMANTIC DOMAIN. Composition rules simply follow from the event propagation constraints and the basic rules for timed automata composition as given in [19]. We identify the following constraints when using the DRE SEMANTIC DOMAIN.

- `Task` to `Task` *connections must be one-to-many.* Events are broadcasted from the source to every dependent. If multiple events are sent to the same task, however, events will be dropped, which is a failure we want to avoid.

- `Task` to `Channel` *connections must be one to many*, which provides a modeling construct - the event channel - to express one-to-many broadcasting and many-to-one event consumptions.
- `Channels` *can only have one dependent.* Since event channel has only one buffer it cannot keep track of the buffer of individual tasks.
- `Channels` *can only have one source.* We have previously allowed a single task to broadcast events to multiple channels. If a task A is connected to an event channel C that channel also receives events from a task B emitting an event from task A will be received by task B, as well, since connections are unidirectional. We therefore disable this modeling construct.

3.8 Dependency and Platform: The Aspects of Scheduling

We propose a platform-based analysis of DRE systems consisting of two major aspects: *dependency*, which describes various relations and dependencies between tasks, and *platform*, which specifies the platform that executes the tasks.

The timed-automata based model checking captures both dependency and platform aspects of the scheduling, and provides both sufficient and necessary conditions for distributed system schedulability. The Scheduler expresses the scheduling algorithm on the underlying platform - decoupled from the tasks and event channels that specify the dependencies. The verification is inherently complex and is most suitable for analysis in a mission-critical resource-constrained embedded domain.

In the embedded domain there is no single method that prevails over others. Testing captures both the platform and dependency aspects of the analysis but can only cover a subset of the possible execution traces. Classic scheduling theory [18, 20] is generally strong in analysis that involves the platform aspect, *e.g.*, offering polynomial execution time for several scheduling problems. Synchronization is often assumed, however, and dependencies between tasks are usually neglected. The analysis of mission-critical DRE systems has to utilize all these methods - schedulability analysis, testing, and model checking - to certify, and validate safe behavior.

3.9 Analysis for Preemptive Scheduling

The DRE Semantic Domain model provides a solid foundation for verifying non-preemptive scheduling on distributed platforms. This model can be extended to handle preemptions by defining a location in the Task model in which the execution time does not progress. This extension, however, turns the model into the stopwatch model of computation that can be expressed using hybrid automata.

Deciding the schedulability of the stopwatch model with execution intervals using timed automata has been shown to be undecidable [21]. Current approaches for stopwatch automata verification include semi-decidable methods for reachability analysis using hybrid automata [22, 23]. Potential directions to offer decidable results include using timed automata for the conservative approximation of the scheduling problem based on time discretization. This approximation is a promising direction to increase the performance of the verification, which is crucial in larger systems. Finding an appropriate time discretization scheme is currently under investigation [24].

4 Model-Based Verification

Developing a DRE system that satisfies multiple QoS properties is a complex constraint-satisfaction problem. To ensure optimal QoS support in practical applications, developers often face hard or even unde-

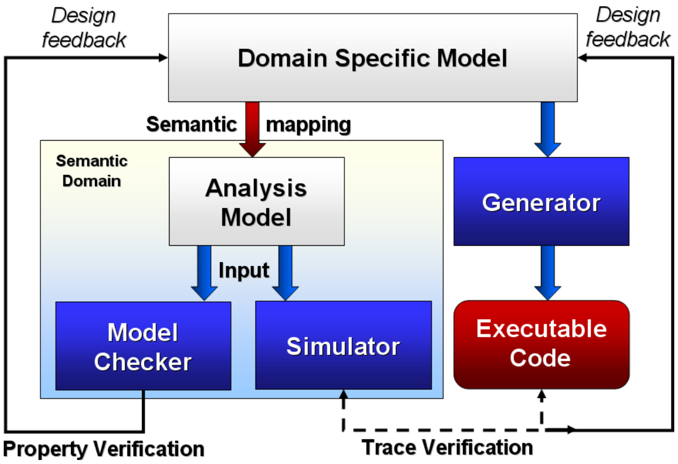

Figure 3: Model Checking using Model Transformations

cidable problems. Despite recent advances in embedded systems' analysis and abstraction techniques [25] the generic verification of production-scale DRE systems is largely unsolved. Time-triggered systems offer better analyzability [26], but increase the implementation complexity and resources needed to satisfy the synchronization.

Figure 3 shows the proposed method for the real-time analysis of distributed embedded systems. This method builds on the results of platform-based design [27] and model-integrated computing (MIC) [28]. Domain-specific modeling languages (DSMLs) play an essential role in the design and analysis process, and can also be used to synthesize executable code, simulations, or documentation. We use model transformations in DREAM to specify the link between the Boeing Bold Stroke modeling language, and the timed automata formal model of computation, as shown in Figure 3. The analysis follows the model checking approach [29] 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 DRE system development by choosing feasible design alternatives.

When designing applications that require support for multiple QoS properties, the analysis often requires multiple tools for the analysis. For example, one tool can be used to verify real-time properties, while simulation can be used to predict the overall power consumption of the system. The DRE SEMANTIC DOMAIN provides a common semantic domain that can capture multiple QoS properties of a generic class of DRE systems, including Real-time CORBA applications. This 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 the simulator can be used to simulate the execution trace that yields this undesired behavior. For example, a model checker may find system deadlocks by checking the formal model of computation generated from the domain-specific application model.

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 analyis in DREAM. This domain has an underlying timed automata semantic domain that can be checked for correctness by model checkers, such as UPPAAL, Kronos or the IF toolset.

4.1 The Graph Transformation

Model transformations play an important role in Dream, where system designers can formalize their transformation from the DRE systems' modeling language to the formal model of computation used for the analysis. The IF toolset presents a similar method for transforming UML [30] into the IF intermediate format. Our method, however, uses domain-specific modeling languages instead of a single generic modeling language. We believe that defining semantics to smaller modeling languages and their composition is more likely to succeed than to define it for a large generic modeling language.

The current Dream design uses the *Generic Modeling Environment* (GME) [31] and *Graph Rewriting and Transformation* (GReAT) [32] tools for the transformation. The implemented graph transformation is an abstract syntax-level transformation, which takes Boeing Bold Stroke models as inputs and generates the corresponding timed automata representation. The developed algorithm creates the set of timed automata, and the corresponding communication channels and global (shared) variables. For example, the array `en[ ]` used to signal the scheduler which tasks are enabled (denoted as the *idle* variable on Figure 2) is defined as a global variable.

The transformation algorithm is relatively straightforward. We generate a *Task timed automaton* for every invocation unit in Bold Stroke and a *Channel timed automaton* for every real-time event channel in Bold Stroke. The graph transformation composes the timed automata representing the Bold Stroke models. We use the Uppaal event channel mechanism to create the links between the elements of the DRE Semantic Domain.

4.2 A Note on Property-Preserving Transformations

We illustrate how the timing properties are preserved by using the definition of timed languages as defined in Section 3.2. We claim that any timed language corresponding to a schedulable (unschedulable) execution trace is accepted (rejected) by the timed automata. Since the Boeing Bold Stroke execution framework does not have a formal specification, we simply illustrate the correctness of the algorithm instead of a proof.

The major advantage of model checking is that it covers every possible execution trace. The timed automata models presented in this paper are a high-level abstraction of the underlying architecture that capture key properties. To keep the system analyzable, however, they abstract out several details, such as computations carried out by the tasks and their implementations.

The Bold Stroke models capture dependencies that are a partial ordering between tasks. We define a timed automaton for each task and specify the same partial ordering by using the composition methods described in Section 3. Each timed automaton is an abstract real-time model of its corresponding task since we define the parameters of the timed automata by using the parameters of the Bold Stroke models. Since the parameters and dependencies are the same, a schedulable execution trace in Bold Stroke will be accepted by the timed automata, while unschedulable traces will be rejected.

4.3 Implementation in GReAT

To illustrate the graph matching pattern we chose a representative rule shown in Figure 4. Both the Bold Stroke and TimedAutomata DSMLs are defined by the GME metamodel that define the abstract syntax of the DSML. We use these metamodels to define the pattern matchings in GReAT.

A rule usually has multiple input and output ports. For example, `Actions` correspond to `TimedAutomata` and `Events` correspond to `Channels`. We want to express that whenever an `Action` publishes an `Event`, the corresponding `TimedAutomaton` should send an event to the corresponding `Channel`.

There are several methods to implement this simple rule. We can define the whole graph matching pattern we are looking for and simply define the pattern to be created in the target model. We try to break up complex pattern rules into simple ones whenever feasible, however, since the performance of the pattern matching is proportional to the “size” of the pattern.

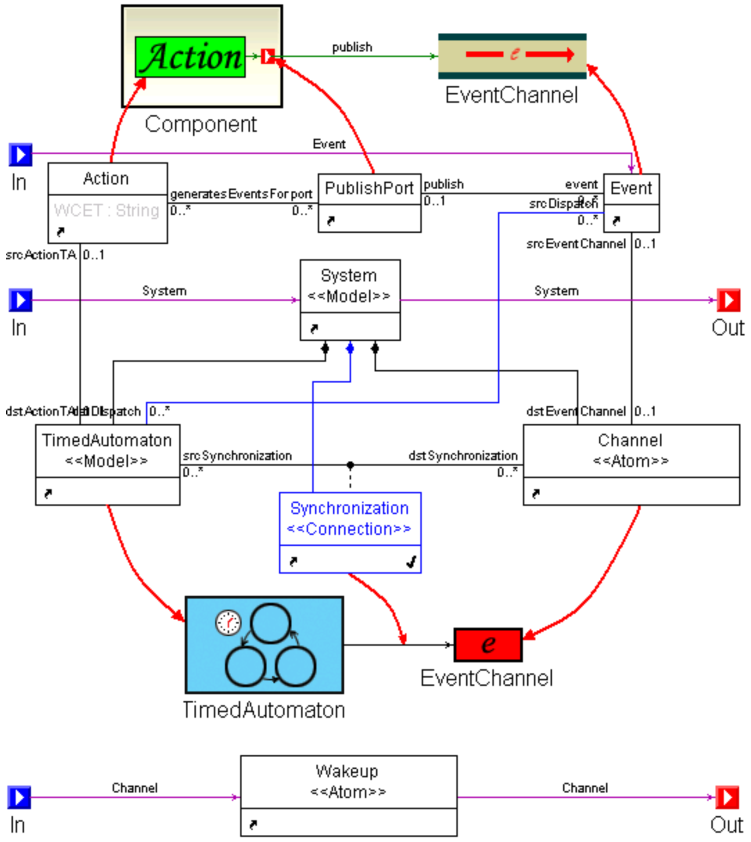

Figure 4: The TA2Channel Rule

In the previous steps we create an empty timed automata model, a `TimedAutomaton` for every event-driven `Action`, and a `Channel` for every `Event` that represents a real-time event channel in the middleware. We have already created the `TimedAutomata` and `Channel` constructs during the previous steps of the transformation. To create the appropriate links between a `TimedAutomaton` and `Channel` we need to find the `TimedAutomaton` in the newly created model corresponding to the `Action` and the `Channel` corresponding to the `Event`.

The transformation rule shown in Figure 4 takes 3 inputs and produces 2 outputs. The input shown on the top of the figure is an `Event`. We define a graph matching pattern for the association between the `Event`, `PublishPort`, and `Action` (we can navigate in both directions on an association). The concrete syntax is shown with curvy arrows in Figure 4. The second input is the whole timed automata model itself. The concrete syntax of these objects is also shown in Figure 4.

We add a `Synchronization` connection between the `TimedAutomaton` and the `Channel` objects. The output of this rule will pass this newly created `System` model to the next rule. The third input, the `Wakeup` object, is simply forwarded to the next rule, which is actually a `Channel` that was renamed to avoid confusion between this and the other `Channels` used for the second input.

4.4 Open-source Dream

The GME and GReAT tools provide the foundation of the model-based verification architecture of Dream. The transformation from the Boeing Bold Stroke modeling language to timed automata is an abstract syntax-level graph transformation specified by the GReAT transformation language. We have recently been released a platform-independent ANSI-C++ open-source implementation of Dream (`dre.sourceforge.net`) to promote the application of analysis tools to resource-constrained DRE systems. The open-source Dream is a lightweight implementation of the GME tool-chain that includes a discrete event simulator - but has a continuous time model - and provides a way for the *runtime analysis* of a generic class of DRE systems. It is therefore able to provide a formal proof for the correctness of a general class of DRE systems.

The open-source Dream tool also has an eXtensible Markup Language XML [33] interface with schema validation for the DRE Semantic Domain. The model transformations are internally represented as template-based data generation and the XML format allows the use of XSL Transformations [34] to the DRE Semantic Domain.

4.5 Verification of the Timed Automata Using Model Checking

To show that the Bold Stroke application is correct, we checked that the corresponding timed automata model is deadlock-free by using the `A[] not deadlock` Uppaal macro. This macro checks that eventually every task does not deadlock, *i.e.*, reach a state from which no transition is possible and time cannot progress. We also need to show that all invocation units meet their deadlines, which requires no additional checking of properties beyond checking for deadlocks.

To reduce the state space, we set the `error` location to be `committed`. By using this constraint in the system we also introduced a nice side-effect, *i.e.*, whenever a timed automaton reaches the `error` location time cannot advance in that automaton. Since we cannot leave that location, the timed automata model will deadlock. If the above reachability macro evaluates to `true`, therefore, we have proved that there are no deadlocks in the system and every action always finishes its execution before the deadline. We also prove that every published event is consumed properly in the system and the event channels operate with limited buffer size.

Additional timed properties can also be checked because all timing properties (such as deadlines, execution times, and dependencies) are captured in the timed automata models. Our model-based Dream verification can be used to verify the correctness of non-preemptive scheduling and pinpoint components that have frame overrun conditions. Our results show that the deadline is not a function of the period and using slower timer components may produce the same properties in the system, thereby allowing better resource allocation and performance gain.

Model checking can express DRE system *behavior*, which enables fine-grained evaluation of QoS properties, such as end-to-end deadlines, graceful degradation and dependability. For example, we can use model checking to show the graceful degradation of a DRE system by increasing the WCETs according to the increased system load. Dependability can be checked by introducing faults in the system models. Model checking techniques thus help evaluate DRE systems in real-world scenarios, as discussed in Section 5.

5 Case Study

To illustrate the model-based verification capabilities of Dream, we examine a case study of a DRE system from the domain of avionics mission computing. Figure 5 shows the component-based architecture of this system, which is built upon the Boeing Bold Stroke real-time middleware described in Section 2. This application is deployed on a non-preemptive multiprocessor platform.

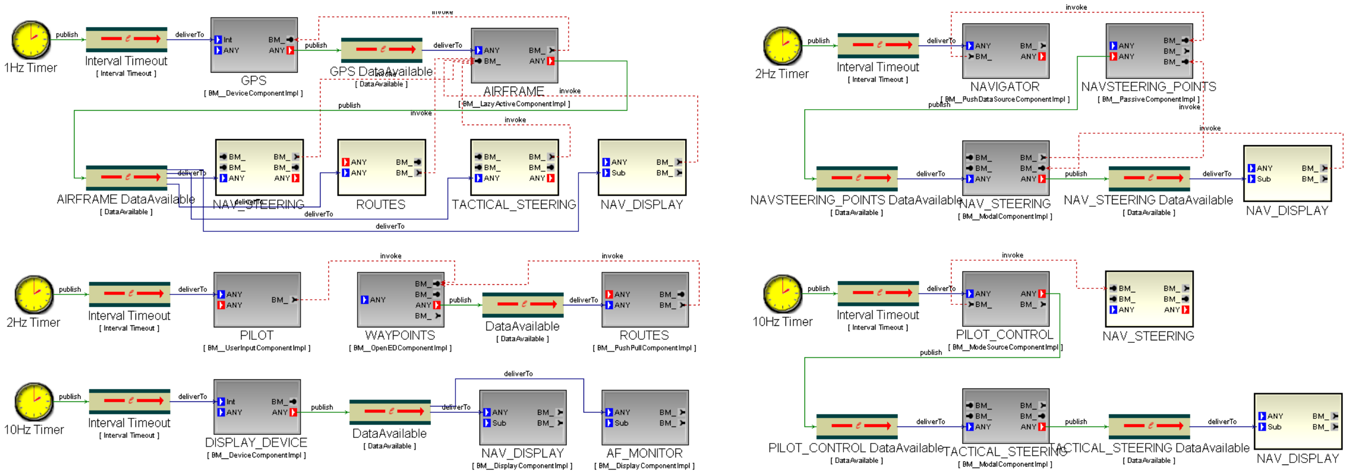
1Hz Timer
Interval Timeout
[Interval Timeout]
GPS
[BM__DeviceComponentImpl]
GPS DataAvailable
[DataAvailable]
AIRFRAME
[BM__LazyActiveComponentImpl]
AIRFRAME DataAvailable
NAV_STEERING
ROUTES
TACTICAL_STEERING
NAV_DISPLAY
2Hz Timer
PILOT
[BM__UserInputComponentImpl]
WAYPOINTS
[BM__OpenEDComponentImpl]
DataAvailable
ROUTES
[BM__PushPullComponentImpl]
10Hz Timer
DISPLAY_DEVICE
[BM__DeviceComponentImpl]
NAV_DISPLAY
[BM__DisplayComponentImpl]
AF_MONITOR
[BM__DisplayComponentImpl]
2Hz Timer
NAVIGATOR
[BM__PushDataSourceComponentImpl]
NAVSTEERING_POINTS
[BM__PassiveComponentImpl]
NAVSTEERING_POINTS DataAvailable
NAV_STEERING
[BM__ModalComponentImpl]
NAV_STEERING DataAvailable
NAV_DISPLAY
10Hz Timer
PILOT_CONTROL
[BM__ModeSourceComponentImpl]
NAV_STEERING
PILOT_CONTROL DataAvailable
TACTICAL_STEERING
TACTICAL_STEERING DataAvailable
NAV_DISPLAY
publish
deliverTo
invoke

Figure 5: The Bold Stroke Application Model

As shown in Figure 5, this application is driven by five `Timer` components deployed on five CPUs. The `GPS` and `AIRFRAME` components are deployed on `CPU_1`. When the `1Hz Timer` component pushes an event the `GPS` component will be notified, and scheduled for execution by the OS (operating system) scheduler. The `GPS` component then pushes an event to the `AIRFRAME` component. The OS scheduler schedules the `AIRFRAME` component for execution, which calls back to the `GPS` component's facet using its receptacle to get the actual data required for execution. The `AIRFRAME` component pushes an event to each of the `NAV_STEERING`, `ROUTES`, `TACTICAL_STEERING`, and `NAV_DISPLAY` components. Since these components are deployed on different processors they are presented in lighter colors in the `1Hz Timer`'s band.

Computations on different processors are driven by their respective timers. Components do not necessarily execute with the timer's rate, however, as seen in the `NAV_DISPLAY` component's case. It is executed more often to serve remote requests than to serve local requests on `CPU_3`. We observe the following key challenges in 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 that are publishing to the same component (*i.e.*, between the event channels that publish events to `AIRFRAME`).

- *Delays.* Communication between processors incur delays in the message propagation. Since the delays are not constant, race conditions may occur when a lower priority task receives an event earlier than a higher priority task, which can result in priority inversion.

- *Composition.* The problems above can be summarized as composition challenges, *i.e.*, *the schedulability of individual threads does not guarantee the overall schedulability of the system.*

5.1 Abstractions Based on the Threading Model

The publish/subscribe architecture used in the the Boeing Bold Stroke execution framework defines two types of mechanisms for data exchange and dependencies.

- Remote method invocations follow conventional two-way function call semantics when a component issues a call from its receptacle to the target component's facet. These two-way facet/receptacle method calls will block if the called process is already executing, which can degrade performance significantly.

Component	CPU	Sub-priority	WCET	Deadline
GPS	CPU_1	VERY_HIGH	21	22
AIRFRAME	CPU_1	HIGH	53	54
PILOT_WAYPOINTS	CPU_2	VERY_HIGH	37	38
ROUTES	CPU_2	VERY_LOW	18	19
NAVIGATOR_NAVSTEERING_POINTS	CPU_3	VERY_HIGH	32	65
NAV_STEERING	CPU_3	HIGH	49	50
DISPLAY_DEVICE	CPU_4	VERY_HIGH	26	41
AF_MONITOR	CPU_4	HIGH	33	34
NAV_DISPLAY	CPU_4	MEDIUM	14	74
PILOT_CONTROL	CPU_5	VERY_HIGH	23	66
TACTICAL_STEERING	CPU_5	HIGH	58	59

Channel	Buffer Size	Worst-case Delay	Best Case Delay
nav_steering_lc	2	0	0
routes_lc	2	0	0
tactical_steering_lc	2	0	0
nav_display_lc	2	0	0
af_monitor_lc	2	0	0
nav_steering_rc	2	2	1
routes_rc	2	2	1
tactical_steering_rc	2	3	1
nav_display_rc1	2	3	1
nav_display_rc2	2	3	1
nav_display_rc3	2	2	1

Table 1: Parameters for the Bold Stroke Application Shown in Figure 5

- Event propagations provide a more efficient asynchronous data flow semantics from event sources to event sinks supported by event channels. The event channel is built on the asynchronous method invocation (AMI) [35] feature of the CORBA specification. The event channel is the implementation of an agent that manages the event passing between tasks. The caller thread A issues the method call on the event channel and resumes its execution, rather than waiting for the called thread B to process the event. When the called thread B finishes the execution it notifies the event channel, which issues the remote method on thread B as thread A's agent.

We assume a time interval for the delivery of the events that is independent for each processor, and is specified by *best case delay* and *worst-case delay*. The event passing between different processors is managed by the *remote* real-time event channel. They are modeled by the `Channel` modeling construct in the DRE Semantic Domain. Race conditions may occur in remote event passing and are an important issue to be solved by analysis. Event passing on the same processor (same address space) is managed by *local* real-time event channels. They are modeled by the `Buffer` modeling construct in the DRE Semantic Domain. In this scenario the ORB delivers the event by invoking local functions - without marshaling/demarshaling the requests. This technique is called a *collocation optimization* [36] and is implemented in most CORBA ORBs, including TAO.

Collocated event passing does not follow the strict event passing semantics since a single thread is used to manage both the event sources and event channels. We therefore assume that the event channel notifies every task before the scheduler is invoked, thus there are no race conditions within a thread. This mechanism helps to enforce fixed priority scheduling for tasks that receive events at the same time and are deployed on the same thread.

Method invocations inherit the QoS execution semantics from the invoking task and do not interact with the runtime scheduler. If the caller and invoked tasks are deployed in the same address space of a processor, the invoked task executes within the same thread - the dispatcher (worker) thread - as the caller task. If the two tasks are deployed on separate machines the ORB uses a thread to transparently forward the method call to a task on the remote machine. This mechanism separates the scheduling of the dispatcher (worker) thread - which schedules tasks invoked by event propagations - and the ORB thread - which simply executes remotely invoked tasks.

The threading model described above provides a way to agressively abstract out remote method calls from the model. For local method calls we simply add the WCET of the called and caller tasks. For remote method calls we also add the worst-case delay of the channel to the WCET of the call chain. This approach assumes that the ORB thread is always ready to serve requests. Although this assumption is overly optimistic in some cases, we can always add more ORB threads or let the ORB's Portable Object Adapter (POA) manage dynamic number of threads in the thread pool.

5.2 Compositional Analysis Using Uppaal

We use the Uppaal model checker tool to check whether various properties hold such as deadlock-freedom, bounded buffer sizes, and whether all deadlines are met. 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) [37], 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 a subset of the Bold Stroke system as modeled in the Uppaal model checker tool. The application consits of 11 task components and 11 event channels, which 5 are local, and used only for buffering. The application is deployed on 5 processors. The `Timer` components are a simple rate generators which publishes events at a predefined rate. We model them using `Timers` in the DRE Semantic Domain.

To satisfy real-time constraints and avoid unnecessary thread spawn delays, the PRISM component middleware requires dedicated threads for each real-time event channel. In the DRE Semantic Domain, however, we can abstract out some of these threads 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. All the event channels satisfy one of the above conditions, except the timer's event channels that have been abstracted out in the model.

The scheduling policies are represented by `Schedulers` in the DRE Semantic Domain. We define 5 schedulers since the Bold Stroke application is deployed on a 5-processor architecture. We illustrate the scheduler of `CPU_4` as shown on Figure 6. The schedulers get more complex according to the scheduling policies. The automatic generation of the models provides a safe way to ensure the correct guard conditions and assignments.

The timed automata model illustrated in 6 corresponding to the Bold Stroke system shown in 5 has been shown to be schedulable using Uppaal. We have checked the system for deadlocks and missed deadlines by using `A[] not deadlock` Uppaal macro. We have checked whether the system operates with finite buffer sizes with the TCTL formula `A[] (`*`Channel`*`.bufferc < `*`Channel`*`.lambdac)`. Uppaal produces a counter-example for invalid properties, which helps identifying the source of undesired behavior. Finally, we checked that eventually every task will execute using the formula `E<> `*`Task`*`.executing`.

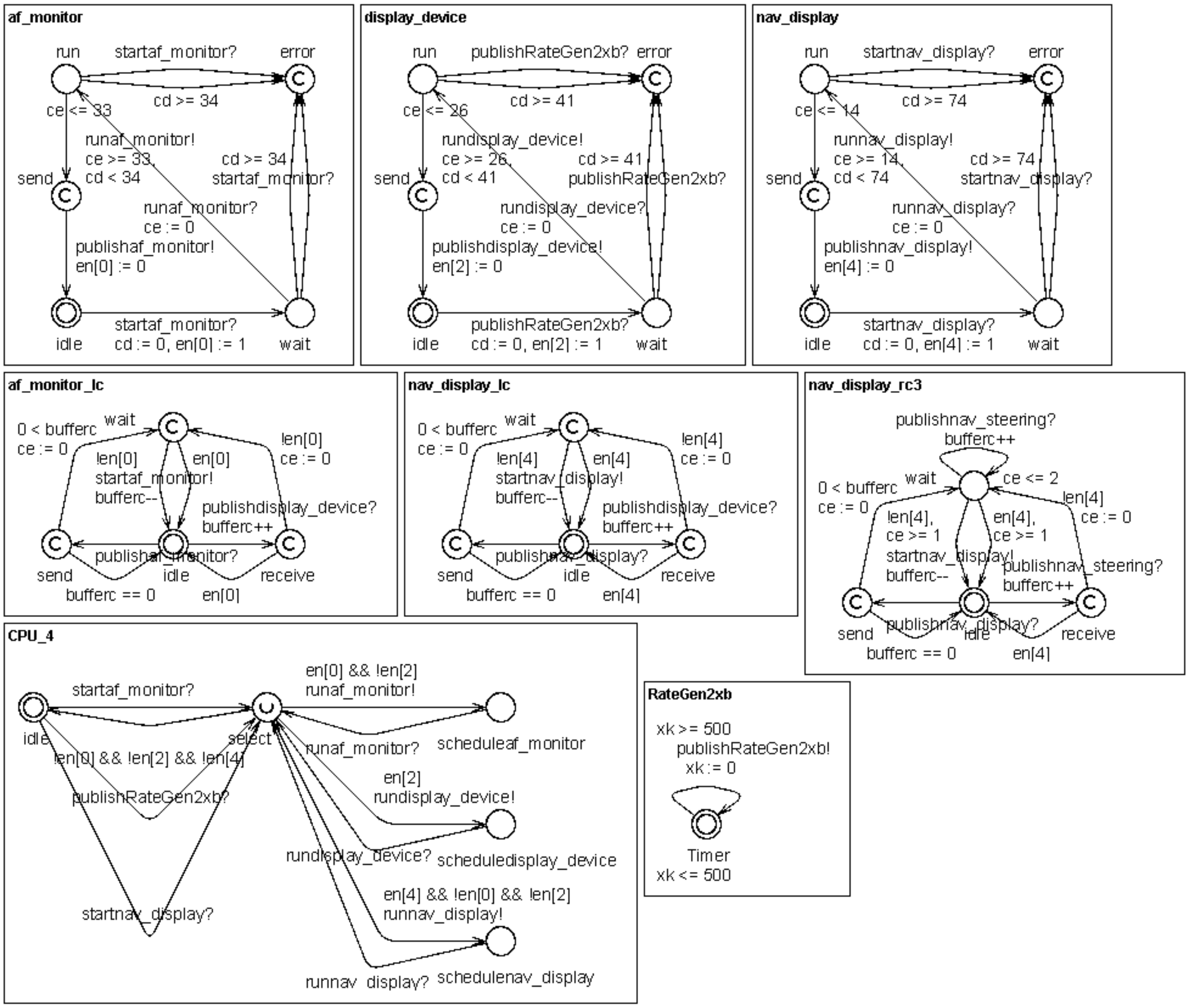

Figure 6: Uppaal Timed Automata Models for CPU_4 (Partial Figure)

The performance of the verification depends largely on the number of non-deterministic branches in the event flow, not the number of components. Uppaal uses powerful reduction techniques for deterministic steps. The properties described above for the system shown in Figure 5 can be checked within 1-2 seconds on a 1.6GHz Pentium 4-M processor with 768 MB memory running the Windows XP OS. Designers must aim to ensure deterministic scheduling and behavior for critical system tasks. Our experiments confirm that the complexity grows exponentially with respect to the state space size. Finding the right abstraction is therefore crucial for tractable verification problems.

6 Related work

This section compares our research on model-based verification using Dream with related work.

Classic schedulability methods. Classic scheduling theory, such as Rate Monotonic Analysis (RMA) [18, 20] or Earliest Deadline First (EDF) [38], is widely used for the schedulability analysis of DRE systems. Synchronization is often assumed, however, and dependencies between tasks are often neglected. Most of the methods provide sufficient, but not necessary conditions for DRE system schedulability.

A holistic method is proposed for distributed preemptive scheduling [39] that provides sufficient conditions for schedulability using a TDMA communication bus. The model is general enough to represent a handful of real-time systems but is not suitable for DRE systems where tasks communicate in an unsynchronized way. In contrast, DREAM is capable of modeling asynchronous communications using the event channel model.

Methods that consider task dependencies are usually extensions of the job shop scheduling problem, which is NP-complete [40]. The Resource Constrained Project Scheduling Problem (RCPSP) is a prominent extension for which the fastest solutions use genetic algorithms and metaheuristics [41].

Other authors [4, 42, 5, 43] have used model-checking techniques and tools for dynamic analysis of DRE systems. The underlying models are variants of the timed automata model. A generic form to analyze scheduling behavior based on the timed automata model was proposed in [4] for single processor scheduling using the Immediate Ceiling Priority protocol and the EDF algorithm. Other methods propose the use of the stopwatch model to verify preemptive scheduling of real-time systems [22, 23]. Contrary to earlier works, the DREAM framework focuses on component-based DRE systems. DREAM provides a conservative approximation built on timed automata [24] to verify preemptive scheduling. This method requires further research before it can be applied directly to DRE systems analysis. In contrast with the manual abstraction of key properties in classical DRE system design, the proposed model-based verification uses automatic model transformations to ensure proper analysis. Our experience developing DRE systems [44] suggests that this aspect of the analysis phase is often overlooked.

A promising way to address timed automata composition using priorities is presented in the IF toolset [12]. Implementations should not use a predefined set of priority levels, however, because it constrains the composition and requires additional guards to express fixed-priority scheduling. The IF toolset presents a method for transforming UML [30] into the IF intermediate format. DREAM complements this work and focuses on the component-based DRE systems domain providing an open-source implementation for design-time and runtime analysis of DRE systems.

Model driven middleware. The FORGE project [45] is a framework for optimization of distributed embedded systems software. DREAM is part of the FORGE project and is tailored to analyze power-aware resource-constrained mobile DRE systems.

The *Component Synthesis using Model Integrated Computing* (CoSMIC) [3] toolkit is an integrated collection of DSMLs that support the development, configuration, deployment, and evaluation of DRE systems based on CIAO, which is an implementation of the CORBA Component Model that is integrated with Real-time CORBA [8]. The CoSMIC tools can be used to specify requirements, compose DRE systems and their supporting infrastructure from the appropriate set of middleware components, synthesize the metadata, collect data from application runs, and analyze the collected data to re-synthesize the required metadata. CoSMIC currently does not support formal analysis of DRE systems therefore it could be applied complementary to DREAM in the design process.

MetaH `www.htc.honeywell.com/metah` is a commercially available domain-specific architecture description language (ADL) for developing reliable, real-time multiprocessor avionics system architectures. A significant set of tools have been protoyped and used in the context of avionics applications. MetaH is a very low level tool, however, since it is largely a language for assembling existing pieces of code. DREAM, in contrast focuses on a high-level abstraction of the system to obtain formal guarantees on system behavior.

Boeing Bold Stroke analysis tools. Several tools have been developed for the model-based design, optimization, analysis and verification of distributed real-time embedded (DRE) systems. The NSF ITR (Information Technology Research), DARPA MoBIES (Model-Based Integration of Embedded Systems), and DARPA PCES (Program Composition for Embedded Systems) projects focus on the *integration* of these tools. Results of the research on Bold Stroke scheduling and verification are the VEST, AIRES, Cadena, and Time Weaver - TimeWiz® tools.

The Virginia Embedded Systems Toolkit (VEST) [46] is a framework designed for the reliable and configurable composition and analysis of component-based embedded systems from COTS libraries. The modeling environment uses the GME [31] tool. VEST applies key checks and analysis but - unlike Dream - 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. Various polynomial-time analysis tasks are supported such as checking for dependency cycles as well as forward/backward slicing to isolate relevant components. It performs real-time analysis [47] using Rate Monotonic Analysis techniques. The authors also present a method [48] to verify the preemptive scheduling of Bold Stroke applications on a single processor. Their method uses constant execution times for tasks and composes the verification results of individual threads/priority bands to obtain a proof for the whole system. The safe composition requires restrictions on the communication between priority bands, otherwise the system load can increase in individual threads thus turning the system unschedulable. In contrast, Dream presents a method to verify non-preemptive scheduling on distributed systems with unrestricted communication model, execution intervals for tasks and delays as well, and also captures the half sync-half async architecture of Bold Stroke by using the event channel model.

The Cadena [49] framework is an integrated environment for building and analyzing CORBA Component Model (CCM) based systems. Its main functionality includes 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. In contrast, Dream represents time explicitly and allows quantitative reasoning.

Time Weaver (Geodesic) [50] is a component-based framework that supports the reusability of components across systems with different para-functional requirements. It supports code generation, as well as automated analysis. Time Weaver also builds a response chain model [20] of the system to verify timing properties. This model is used by real-time analysis tools, such as the TimeWiz® model-checker, to build a task set that can be analyzed with RMA techniques. In contrast, Dream does not assume a synchronized system and captures task dependencies explicitly.

7 Concluding Remarks

This paper presents the Dream model-based verification and analysis framework. Dream uses formal methods to capture and verify properties of non-preemptive, event-driven component-based DRE systems that use the publish/subscribe communication pattern. Its verification is automatic, exhaustive, and capable of producing counter-examples that help pinpoint sources of undesired behavior.

We have applied Dream to Boeing Bold Stroke avionics mission computing platform, which is representative of state-of-the-practice DRE systems based on QoS-enabled component middleware. The Bold Stroke event-driven component-based DRE system platform is built atop the TAO Real-time CORBA ORB and the TAO Real-time Event Service. The goal of this paper was to verify QoS properties that express the *behavior* of this DRE system, such as end-to-end deadlines, graceful degradation, and dependability.

Our results showed that Dream captures the reactive behavior, as well as the non-determinism present in DRE systems like Bold Stroke. We also demonstrated that timed automata can represent component interactions and asynchronous event passing allowing the verification of quantitative dense time properties.

Key challenges for the DREAM framework include the integration of frameworks tailored towards model-driven development, such as CoSMIC [3], integration of various model checkers such as Spin [51], and the IF toolset, and the use of heuristics for optimizations in resource-constrained mobile environments. Extending the DREAM framework is a key part of our future work, which focuses on expressing the formal, heterogeneous composition of semantic domains to support better and more robust DRE systems development. The open-source DREAM implementation is available for download at `dre.sourceforge.net`.

References

[1] Object Management Group, "CORBA Component Model," 2002. [Online]. Available: http://www.omg.org

[2] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service," in *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM Press, 1997, pp. 184–200.

[3] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, "Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications," *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2005 (to appear).

[4] T. Gerdsmeier and R. Cardell-Oliver, "Analysis of Scheduling Behaviour using Generic Timed Automata," vol. 42, 2001.

[5] C. Ericsson, A. Wall, and W. Yi, "Timed Automata as Task Models for Event-Driven Systems," in *Proceedings of RTSCA '99*, 1999.

[6] D. C. Sharp and W. C. Roll, "Model-Based Integration of Reusable Component-Based Avionics Systems," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[7] 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).* IEEE Computer Society, 2003, pp. 75–82.

[8] *Real-time CORBA Specification*, OMG Document formal/02-08-02 ed., Object Management Group, Aug. 2002.

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

[10] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, Oct. 1997.

[11] 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.* Springer-Verlag New York, Inc., 1996, pp. 208–219.

[12] 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*, pp. 237–267, Sep 2004.

[13] D. C. Schmidt, A. Gokhale, T. H. Harrison, and G. Parulkar, "A High-Performance Endsystem Architecture for Real-Time CORBA," *IEEE Communications Magazine*, vol. 14, no. 2, 1997.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns.* New York: Wiley & Sons, 1996.

[15] 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.

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

[17] M. Deshpande, D. C. Schmidt, C. O'Ryan, and D. Brunsch, "Design and Performance of Asynchronous Method Handling for CORBA," in *Proceedings of Distributed Objects and Applications (DOA)*, October/November 2002.

[18] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[19] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[20] M. H. Klein, T. Ralya, B. Pollak, and R. Obenza, *A Practitioners' Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.

[21] P. Krčál and W. Yi, "Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata," in *Proc. of TACAS'04, Barcelona, Spain.*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer–Verlag, 2004, pp. 236–250.

[22] J. McManis and P. Varaiya, "Suspension automata: A decidable class of hybrid automata," in *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification.* London, UK: Springer-Verlag, 1994, pp. 105–117.

[23] Y. Abdeddaïm and O. Maler, "Preemptive job-shop scheduling using stopwatch automata," in *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* London, UK: Springer-Verlag, 2002, pp. 113–126.

[24] G. Madl and S. Abdelwahed, "Model-based Analysis of Distributed Real-time Embedded System Composition," in *EMSOFT 2005*, 2005.

[25] 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.* New York, NY, USA: ACM Press, 2002, pp. 58–70.

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

[27] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An Integrated Electronic System Design Environment," *Computer*, vol. 36, no. 4, pp. 45–52, 2003.

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

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

[30] J.Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual," 1998.

[31] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, and J. Sprinkle, "Composing Domain-Specific Design Environments," *Computer*, pp. 44–51, Nov 2001.

[32] A. Agrawal, G. Karsai, and A. Ledeczi, "An End-to-End Domain-Driven Development Framework," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct 2003.

[33] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Thid Edition)," 2004.

[34] ——, "XSL Transformations (XSLT) Version 1.0," 2004.

[35] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference.* ACM/IFIP, Apr. 2000.

[36] N. Wang, D. C. Schmidt, K. Parameswaran, and M. Kircher, "Applying Reflective Middleware Techniques to Optimize a QoS-enabled CORBA Component Model Implementation," in *24th Computer Software and Applications Conference.* Taipei, Taiwan: IEEE, Oct. 2000.

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

[38] G. C. Buttazzo, "Rate Monotonic vs. EDF: Judgment Day," *Real-Time Systems*, vol. 29, pp. 5–26, January 2005.

[39] K. Tindell and J. Clark, "Holistic Schedulability Analysis for Distributed Hard Real-Time Systems," *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, vol. 40, pp. 117–134, 1994.

[40] J. Blazewicz, J. Lenstra, and A. R. Kan, "Scheduling subject to resource constraints: Classification and complexity," *Discrete Applied Mathematics*, pp. 11–24, 1983.

[41] S. Hartmann and R. Kolisch, "Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem," *European Journal of Operations Research*, pp. 394–407, 2000.

[42] S. Bradley, W. Henderson, and D. Kendall, "Using Timed Automata for Response Time Analysis of Distributed Real-Time Systems ," in *24th IFAC/IFIP Workshop on Real-Time Programming WRTP 99*, 1999, pp. 143–148.

[43] V. A. Braberman and M. Felder, "Verification of Real-Time Designs: Combining Scheduling Theory with Automatic Formal Verification," in *Software Engineering-ESEC/FSE 99*, 1999, pp. 494–510.

[44] C. D. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt, "Integrated Adaptive QoS Management in Middleware: An Empirical Case Study," *Journal of Real-time Systems*, vol. 24, 2005.

[45] R. Cornea, N. Dutt, R. Gupta, I. Krueger, A. Nicolau, D. Schmidt, and S. Shukla, "FORGE: A Framework for Optimization of Distributed Embedded Systems Software," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.

[46] 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.

[47] 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.

[48] Z. Gu and K. Shin, “Model-checking of component-based real-time embedded software based on corba event service,” in *IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, 2005.

[49] 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.

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

[51] G. J. Holzmann, *The SPIN model checker: Primer and reference manual.* Addison Wesley, 2004, hOL g 03:1 1.Ex.