Reusable Models for Timing and Liveness Analysis of Middleware for Distributed Real-Time and Embedded Systems *

Venkita Subramonian, Christopher Gill
CSE Department, Washington University
St. Louis, MO, USA
{venkita,cdgill}@cse.wustl.edu

César Sánchez, Henny B. Sipma
CS Department, Stanford University
Stanford, CA, USA
{cesar,henny}@cs.stanford.edu

ABSTRACT

Distributed real-time and embedded (DRE) systems have stringent constraints on timeliness and other properties whose assurance is crucial to correct system behavior. Formal tools and techniques play a key role in verifying and validating system properties. However, many DRE systems are built using middleware frameworks that have grown increasingly complex to address the diverse requirements of a wide range of applications. How to apply formal tools and techniques effectively to these systems, given the range of middleware configuration options available, is therefore an important research problem.

This paper makes three contributions to research on formal verification and validation of middleware-based DRE systems. First, it presents a reusable library of formal models we have developed to capture essential timing and concurrency semantics of foundational middleware building blocks provided by the ACE framework. Second, it describes domain-specific techniques to reduce the cost of checking those models while ensuring they remain valid with respect to the semantics of the middleware itself. Third, it presents a verification and validation case study involving a gateway service, using our models.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Model Checking

General Terms: Verification, Design.

Keywords: Middleware, Timed Automata.

1. INTRODUCTION

Significant research over the past decade has made middleware more customizable through the use of pattern-oriented software frameworks [13, 12]. Although this effort has made middleware solutions suitable for a wider range of applications, managing the resulting multiplicity of customization options has become an increasing concern. As is shown in Figure 1, abstractions from higher

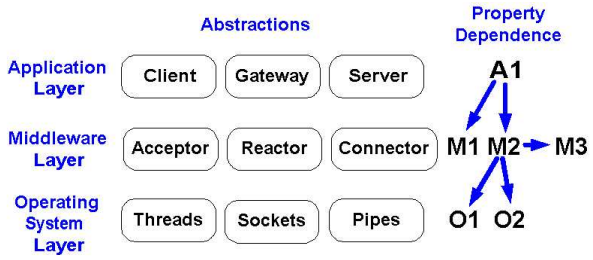


Figure 1: Layers, Abstractions, and Properties

layers of the system software architecture build upon abstractions from lower layers. This establishes an implicit dependence of higher level properties on lower level properties, which in turn makes the formal verification of application-level requirements more difficult in middleware-based systems.

As Figure 1 illustrates, application-level properties such as the timely return of a remote method invocation (A1), may depend on middleware-level properties such as delays introduced by different strategies for establishing and re-using connections with remote endsystems (M1) or for delivering the result of the remote invocation to the application (M2). Furthermore, middleware-level properties such as the delay in delivering the result of the remote invocation to the application (M2) may depend on other middleware properties such as the strategy used to wait for the reply from the remote invocation (M3) and on OS-level properties such as whether threads with the same priority are scheduled with round-robin or run-to-completion semantics (O1) and whether other threads are holding resources needed by the thread that will return the result to the application (O2).

We have focused previously on the design and proof of protocols to enforce application-level properties that depend on other properties at the application, middleware and OS levels. For example, we have developed [22] and optimized [21] thread allocation protocols that can provably avoid deadlock in middleware-based distributed real-time and embedded systems whose 2-way remote method invocation call graph is known. While this approach serves to advance the state of the art in both middleware design and distributed systems theory, careful effort is required to design and prove an enforcement protocol for each property of interest. The research presented in this paper complements the design and proof of en-

*Research supported in part by: NSF CAREER award CCF-0448562 (WUSTL); NSF grants CCR-01-21403, CCR-02-20134, CCR-02-09237, CNS-0411363, and CCF-0430102, ARO grant DAAD19-01-1-0723, and NAVY/ONR contract N00014-03-1-0939 (Stanford).

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

forcement protocols for system properties by establishing a formal foundation that can be re-used effectively to check properties for which enforcement protocols have not yet been developed. Our approach is also useful when existing enforcement protocols do not consider new influences, such as those introduced by other enforcement protocols or by new middleware or OS features.

Our approach offers the following benefits beyond those offered by other related work, which we discuss in Section 7: (1) our reusable, detailed and executable models of middleware building blocks can be composed to model different kinds of middleware, and then can be composed with different application models; (2) the timing and liveness properties of these building blocks can be verified with suitable precision; (3) our models offer a more rigorous and formal representation of detailed middleware engineering expertise that is currently represented as *patterns* [24]; (4) with these resuable low-level models and verification techniques, the extent to which systems must be "over-designed" can be reduced due to greater insight into the possible behaviors of middleware-based systems.

The rest of this paper is structured as follows. Section 2 presents a detailed system model and states the research problem this paper addresses. Section 3 describes the middleware architecture that is captured by our models. Section 4 discusses challenges and solution approaches for modeling concurrent object middleware using the IF tool set. Section 5 discusses domain-specific state space optimizations to allow tractable checking of the models we have developed. In Section 6, we present a case study of scenarios involving a broader set of middleware concurrency and interaction strategies, which in turn affect system timing and liveness properties. Section 7 describes related work, and Section 8 offers concluding remarks.

2. SYSTEM MODEL AND PROBLEM

In this section, we formally describe some of the canonical building blocks [24] used in the construction of real-time middleware and their interactions. We also describe the methodology that we use to model and verify real world examples (see section 6) using middleware built from these canonical building blocks. Although our approach is similar to the approaches used in modeling and verifying component interactions [11] in the context of a component model like CORBA Component Model [29], the key difference from those approaches is the granularity of the elements that interact. The interactions between the middleware building blocks (*e.g.*packet arrival through an interprocess communication channel that can result in triggering a return from a select OS system call, reactor making an upcall to an event handler) are less explicitly structured and more diverse than standard component models (*e.g.*interaction via ports and event channels) and hence must be captured by models that incorporate state transition and timing semantics at a finer granularity than component models. We need the full power of timed automata [1] for capturing the interaction semantics of these middleware building blocks.

Our system model can be expressed as a 6-tuple $\{E, H, I, R, A, \theta\}$, consisting of the following elements:

- E is a set of *events* denoting relevant asynchronous changes in the system's state, such as the expiration of a timer, the arrival of a network packet, or a transport-layer buffer becoming available for writing.
- H is a set of event *handlers*, which perform application-specific processing when system events are dispatched to them.
- I is a set of *interaction* channels, such as sockets and timer registration interfaces, which trigger events as a result of actions performed on them.
- R is a set of *reactors*, which dispatch events to event handlers by invoking event-specific handler methods.
- A is a set of *actions* performed on event handlers, interaction channels, and reactors – such as registering an event handler with a reactor, dispatching an event to an event handler, sending data over a socket, or waiting in a reactor for events to occur.
- θ is a set of endsystem *threads* – actions within a thread are performed sequentially, while actions in different threads can be performed concurrently.

Note that some categories of events (*e.g.*, the return of a thread from a method call) and actions (*e.g.*, invoking a method call) could apply to multiple instances and kinds of system elements. Furthermore, a given event or action can be performed repeatedly. To avoid ambiguity, we assume that every event and every action is identified uniquely, and that each occurrence of a given event or action is indexed uniquely across the entire system. We also assume that each occurrence of an event is instantaneous, while each occurrence of an action has a (possibly different) non-zero temporal duration, and the initiation and completion of each action are represented by distinct events in our system model.

The dynamic interactions between these elements capture formally the relevant parts of the system behavior. These interactions can be transformed into formulas in temporal logic using the relations described below. These formulas can then be proved using formal verification techniques, such as model checking, abstract interpretation, and deductive methods. Sections 4 and 5 describe how these relations can be mapped to the IF model checker, while Section 6 presents a proof using the IF tool. We envision a development cycle where the translation (and its validation) to this formal model is performed by the component developer. Some of these translations and formal proofs can be reused across scenarios, and are composable to form larger proofs specific for the particular application.

Static relations. We first express several static relations in our system model, which hold for the entire system lifetime. These relations partition actions according to the system elements on which the actions can be performed, and partition threads into reactor-specific thread pools:

- $\alpha_H : H \rightarrow 2^A$. The set of actions that can be taken on event handler h is given by $\alpha_H(h)$.
- $\alpha_I : I \rightarrow 2^A$. The set of actions that can be taken on interaction channel i is given by $\alpha_I(i)$.
- $\alpha_R : R \rightarrow 2^A$. The set of actions that can be taken on reactor r is given by $\alpha_R(r)$.
- $threadpool : R \rightarrow 2^\theta$. The set of threads assigned statically to reactor r is given by $threadpool(r)$, with each thread assigned to exactly one reactor, and at least one thread assigned to each reactor. We say that two threads are *local* to reactor r if both are assigned to that same reactor. We say that two threads are *remote* if they are assigned to different reactors.

Temporal relations. We use non-negative real number domain T to denote time, and express several temporal relations in our system model that are useful for the analysis of system timing and liveness properties:

- $registered : E \times I \times R \times T \rightarrow 2^H$. The set of event handlers registered for event e on interaction channel i in reactor r at time t is given by $registered(e, i, r, t)$.
- $active : R \times T \rightarrow 2^E$. The set of events that have arrived at reactor r but have not been dispatched to event handlers at time t is given by $active(r, t)$.

- $ready : E \times R \times T \rightarrow 2^I$. The set of interaction channels for which event e is active in reactor r at time t is given by $ready(e, r, t)$, and a single event-specific action, such as one read from a socket for a "data ready" event, can be taken on a ready channel without blocking the thread in which that action is taken.
- $dispatched : R \times T \rightarrow 2^\theta$. The set of threads in $threadpool(r)$ that are currently in use to dispatch events to event handlers in reactor r, and thus are not available to dispatch other events from $active(r, t)$ at time t is given by $dispatched(r, t)$.
- $blocked : R \times T \rightarrow 2^\theta$. The set of threads in $threadpool(r)$ that have taken blocking actions that will only unblock and allow the thread to continue when a specific event occurs is given by $blocked(r, t)$. Note that for some scenarios, such as a thread scheduling a timer and then blocking on the timer's expiration, unblocking will not depend on an action being performed in another thread; for other scenarios, such as a thread performing a blocking read on a socket, an event to trigger unblocking must be generated by an action taken by another (possibly remote) thread.
- $deadline : \mathcal{N} \times E \rightarrow T$. The time by which the n^{th} occurrence of event e must occur to preserve correctness is given by $deadline(n, e)$. Event occurrences without timing constraints are given a deadline of ∞.
- $occurred : \mathcal{N} \times E \rightarrow T$. The time at which the n^{th} occurrence of event e happened is given by $occurred(n, e)$.
- $live : R \times T \rightarrow 2^\theta$. The set of threads assigned to reactor r within each of which at least one action occurs after time t is given by $live(r, t)$.
- $arrival\_time : \mathcal{N} \times E \times R \times I \rightarrow T$. The time of arrival of the nth occurrence of event e on channel i at reactor r is given by $arrival\_time(n, e, r, i)$. Event occurrences are numbered globally rather than by interaction channel, and if the nth occurrence of an event happened in a different channel than i (or did not happen at all) then the time returned would be ∞.
- $dispatch\_time : \mathcal{N} \times E \times R \times I \rightarrow T$. The time of dispatch of the nth occurrence of event e on channel i by reactor r to the appropriate event handler is given by $dispatch\_time(n, e, r, i)$. If the nth occurrence of event e happened in a different channel than i (or did not happen at all), then the time returned would be ∞.

Problem definition. Our approach hinges on the idea that interference occurs when the actions taken by endsystem threads can affect each other in ways that produce adverse consequences for the system's specified constraints. In this research, we address the specific problem of detecting interference in which threads' actions on reactors, event handlers, and interaction channels in the endsystem middleware can cause violations of application-specific timing and liveness constraints.

Interference. We analyze two forms of interference with timeliness and liveness constraints: blocking delays and exhaustion of threads in a reactor thread pool.

- $blocking\_delay : \mathcal{N} \times E \times R \times I \rightarrow T$. The blocking delay for the nth occurrence of event e is given by the interval between its arrival at a reactor r on channel i and its dispatch to an event handler, $blocking\_delay(n, e, r, i) = dispatch\_time(n, e, r, i) - arrival\_time(n, e, r, i)$. If the nth occurrence of e happened in a different channel and/or reactor than those given to the $blocking\_delay$ function then the return value would be 0.
- $threads\_exhausted : R \times T \rightarrow \{true, false\}$. The threads in the thread pool of a reactor r are exhausted at time t if $|blocked(r, t)| = |threadpool(r)|$.

Our analysis depends both on (1) the specific constraints given and (2) how different middleware mechanisms shape different forms of interference with those constraints. We model the constraints as temporal logic statements and model the middleware mechanisms as timed automata. We then use model checking to evaluate whether or not the constraints are satisfied. Specifically, we search for states of the system in which two particular kinds of constraint violations appear: *missed deadlines*, which are timing constraint violations that can occur even when liveness is preserved, and *deadlocks* which are liveness constraint violations that usually also lead to timing constraint violations in subsequent system states. Checking for a missed deadline can be done using our system model by comparing the time at which the nth occurrence of event e happened, to the deadline for that occurrence of the event: $occurred(n, e) > deadline(n, e)$. Deadlocks can be detected using our system model by determining whether or not we reach a state with global time t after which no further action will be taken by any of a reactor r's assigned threads : $|live(r, t)| = 0$. Note that it is not sufficient to check whether or not all threads in a reactor are blocked: $|blocked(r, t)| = |threadpool(r)|$ says only that no actions can be taken by the threads assigned to reactor r from time t until a subsequent occurrence of an event (*e.g.*, due to an action in a remote thread) causes one of those threads to unblock, and only indicates deadlock if no such event occurs after time t.

When a state containing a constraint violation is reached, the model checker then can produce a trace of the system states that led up to that constraint violation. By examining these traces and correcting the particular patterns of interference they reveal, we can remove design and implementation errors, and also gain insights into designing new enforcement protocols to prevent or avoid constraint violations.

3. MODELING ARCHITECTURE

To be able to verify the correctness of different middleware configurations in the context of each specific application, we have developed detailed and formal models of common middleware building blocks found in the widely used ACE [13] framework. We have modeled reactors, thread pools, event handlers, interaction channels, and other middleware building blocks which can be composed and checked rigorously to evaluate timing and liveness properties in each particular application and its supporting middleware configuration.

Figure 2 shows our modeling architecture, which we have implemented in the context of the IF tool set [3]. We use the IF notation to specify our fine-grained models as *processes* (automata) that run in parallel and interact through shared variables and asynchronous signals. The behavior of these processes is represented formally in IF as *timed automata with urgency* [2] and the semantics of a system modeled in IF is the Labeled Transition System (LTS) obtained by interleaving the executions of its processes.

Our models are divided into three layers: (1) models of network/OS level abstractions such as channels for interprocess communication; (2) models of middleware building blocks such as reactors; and (3) models of application functionality implemented in the form of event handlers. The models themselves are executable in the IF environment and can be model-checked against system property specifications. The unshaded rectangular boxes shown in Figure 2 are modeled using timed finite state automata specified in the IF language. The shaded rectangular boxes shown in Figure 2

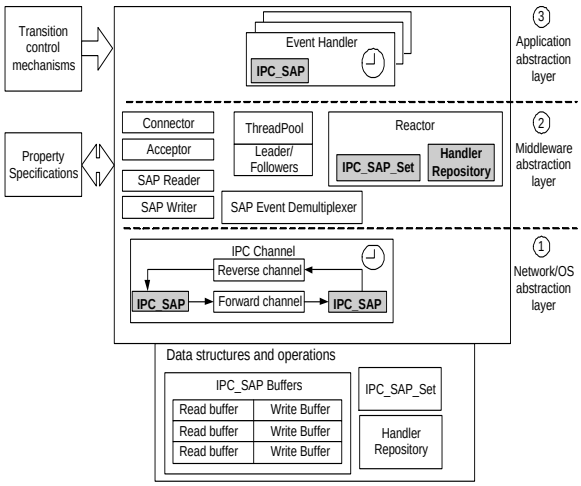


Figure 2: Middleware Modeling Architecture

are data structures that are shared by the different automata in the models. Automata with timed transitions are shown by timer icons in Figure 2.

Network/OS abstraction layer. At the lowest architectural layer we model inter-process communication (IPC) mechanisms, such as sockets, pipes, FIFOs, and message queues, as IPC channels. An IPC channel has two Service Access Points (SAPs), for convenience called the left-hand-side SAP (lhs-SAP) and the right-hand-side SAP (rhs-SAP). Each SAP has a read-buffer and a write-buffer associated with it. The read-buffer is used by the SAP to receive any data sent to it from another SAP and the write buffer is used to send data from that SAP to another SAP. Each SAP has a unique handle associated with it and this handle is used as an index in the IPC channel collection data structure to access the data buffers associated with that SAP.

An IPC channel is bidirectional. It is modeled as two data-transfer automata, one for the forward direction, and one for the reverse direction. The forward channel automaton waits for data to be enqueued on the write-buffer of the lhs-SAP and transfers it to the read-buffer of the rhs-SAP. The reverse channel automaton waits for data to be enqueued on the write buffer of the rhs-SAP and transfers it to the read-buffer of the lhs-SAP. These forward and reverse channel automata also can be parameterized with appropriate propagation delays, if needed.

Middleware abstraction layer. Above the network/OS layer is the middleware layer, where we model abstractions of semantically rich middleware building blocks. Each middleware primitive is modeled so that the behavior seen when the model is executed closely adheres to that of the actual implementation. This faithful modeling of the middleware primitives in turn results in high-fidelity models of higher-level middleware services, obtained by composing these primitive models. To support such faithful modeling, we have developed data structures like the event handler repository used by the reactor to store mappings between a Service Access Point (SAP) and the handler associated with that SAP. This table is populated whenever an event handler is registered with a reactor.

Application abstraction layer. Our models encapsulate application functionality using event handlers, as is customary when developing ACE-based applications in practice. Each event handler reads data from or writes data to IPC channels, which in turn model interactions between different event handlers. The computation performed by an event handler can be modeled as a (potentially complex) automaton, or may be abstracted away and represented by a single transition guarded by a constraint on a timer variable to delay its execution completion event as necessary.

Property specifications for verification. In the IF tool set, observable system properties can be specified by *observers* [3]. These observers are represented by timed automata; they are executed at each step of the labeled transition system (LTS) that is generated from the composed system model before an enabled transition is selected. To facilitate specification, IF provides observer constructs for a variety of events in a system including forking a new process, output events, and input events. In general an observer records an abstraction of the actions and interactions of other automata and also can be used to control the model's execution.

4. MODELING CONCURRENT OBJECTS

Although our goal is to model systems having multiple communicating threads, each of which executes actions including object method calls, the distinction between an object and a thread is not known to the IF model checker. Despite its lack of direct representations for objects and threads, IF proved to be the best suited tool set for our middleware modeling and analysis needs: while Bogor [20] provides native support for objects and threads, IF offers direct support for reasoning about explicit timing, which Bogor does not provide. This decision required us to keep track of the distinctions between threads and objects in the models themselves. In this section we describe the techniques we developed to represent object-oriented concurrent DRE systems in terms of processes (automata) and their interactions in IF.

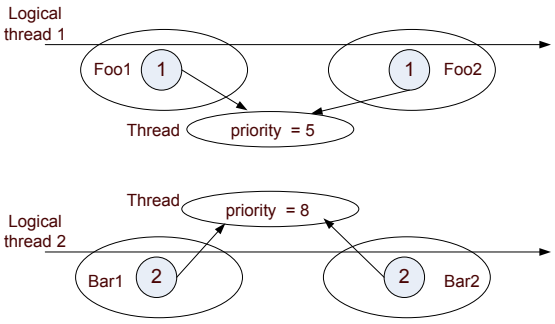
4.1 Modeling Object Interactions

To model object-oriented concurrent systems in IF, each method call must be represented by a separate process, because multiple simultaneous calls can be made to the same object (method), whose computations may interfere with each other. If each object method were modeled by a single process, all calls to this method would be implicitly assumed serialized. This does not, however, correctly represent actual system behavior of, for example, multiple threads in a reactor. We therefore model a method call from one object to another object by having the caller process create a new process for the callee method and send a signal to the new process to start its execution [10]. Upon completion of execution the callee process sends a signal back to the parent (caller) process and stops, thus deleting itself.

4.2 Modeling Threads

In IF, it is the responsibility of the model developer to represent explicitly, in the model itself, the idea of a thread of control flowing through multiple objects as part of a chain of object method invocations. For example, Figure 3(a) shows a *logical* thread that represents a flow of control from one object to another object (Foo1 to Foo2, Bar1 to Bar2), with both of these objects modeled as IF processes. To model a distinct thread flow of control, we developed the concept of a *thread id* maintained by each IF process. The *thread id* is a reference (of type pid in IF) to a unique instance of an IF process of type Thread. Note that the Thread automaton has been developed as part of this research and it is not a built-in feature in IF. The thread automaton serves to record the real-world thread context under which the IF process is executing.

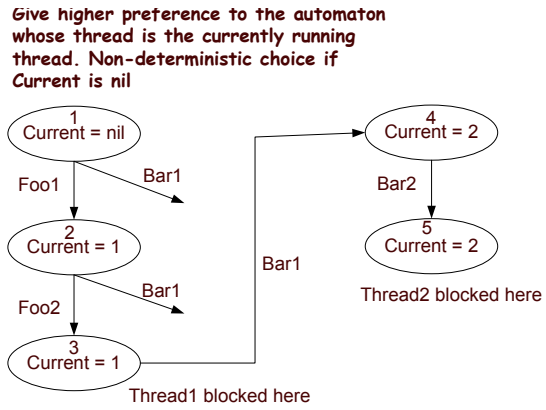
To represent concurrency accurately, any thread in the actual system should be modeled by creating a Thread automaton in our models. When we model an object method invocation, the thread



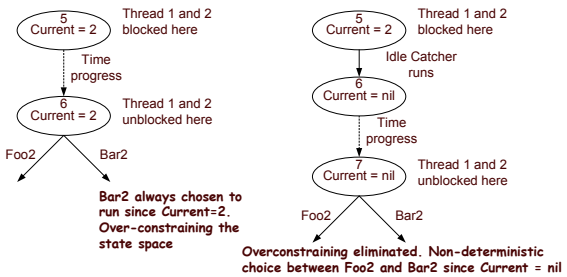
(a) Thread ID Propagation and Scheduling

```
thread_schedule: pid1 < pid2
  if
    pid1 instanceof Foo1 and
    pid2 instanceof Bar1 and
    ({Foo1}pid1).threadid <>
    ({Bar1}pid2).threadid and
    ({Thread}(({Foo1}pid1).threadid)).prio <
    ({Thread}(({Bar1}pid2).threadid)).prio )
```

(b) IF Priority Rules to Model Thread Scheduling



(c) Run-to-Completion Semantics for Two Threads



(d) Idle Catcher

Figure 3: Modeling Thread Semantics in IF

context (represented by a unique thread id) under which that invocation is made is carried over from the caller to the callee object. To propagate the thread context, we use an IF observer [3]. IF observers record abstractions of the actions and interactions of other automata. IF provides observer constructs for a variety of events in a system including forking a new process, output events, and input events. The IF model checker ensures that all enabled IF observers are run between any two *labeled transition system* (LTS) steps. Our thread context propagation observer runs and updates the thread context of a destination process of an IF signal to be the same as the thread context of the source process.

4.3 Modeling Thread Priority Scheduling

In our approach, a Thread automaton with a user-defined state variable named priority can be instantiated to restrict concurrency in execution of the models, just as a priority can be used in OS thread scheduling to reduce interleavings of threads actions in the actual system. Since the Thread automaton is not a built-in construct in IF, a mechanism by which we inform the model checker about the priorities of logical threads is needed, so that the model checker can give preference to transitions that are executing under the context of a higher priority thread over those under the context of a lower priority thread. We use the priority rules feature in IF to specify the priority ordering among different automata interactions.

Figure 3(b) illustrates how we use the priority rules feature in IF to achieve thread scheduling in our models. This IF priority rule states that between two automata of IF process types Foo1 and Bar1, if the thread contexts of these two automata are different, then choose the automaton with a thread id that points to a Thread automaton with a higher value for the priority state variable (in this example, Bar1 is chosen over Foo1).

4.4 Modeling Run-to-Completion Semantics

In the previous section, we discussed how to model priority based scheduling in IF, where we dealt mainly with modeling threads with *different* priorities. In real-time systems, it is very common to use the SCHED_FIFO scheduling mechanism to reduce context switching between real-time threads of the *same* priority. We use a similar technique to control interleavings between the execution of automata within different logical threads of the same priority.

In our models, each logical thread of control runs across multiple IF automata until the thread blocks, or is preempted by a higher priority thread - only then can another thread start running. In the IF model, this translates to the notion of automata in the same thread context executing in sequence until there are no enabled transitions in the group of automata running under that thread context.

To realize run-to-completion semantics in IF, we developed a combination of techniques: (1) keeping track of the currently running thread id as part of the state space; (2) performing thread context propagation from a caller object to callee object; and (3) using an idle catcher to reset the currently running thread when none of the processes in our model have any enabled transitions. For the purpose of this discussion, we assume that Thread1 and Thread2 have the same value for their priority state variables.

Currently running thread context. A globally accessible state variable Current is used to record the thread context which is currently running. Each transition in every automaton in the model updates this global variable with the locally stored thread context under which that automaton is running. If multiple automata are enabled, then only one automaton is selected by the model checker to update the value of Current. Any IF process P whose thread context is the same as the currently running thread will get preference to any IF process Q whose thread context is not the same as the

currently running thread, provided the threads for P and Q have the same priority. This policy is expressed in IF using a combination of IF priority rules. Note that if there is no currently running thread context, then we allow appropriate nondeterminism in the model. For example, Figure 3(c) illustrates an execution sequence where in state 1, the Foo1 automaton is chosen non-deterministically over Bar1 since the value of Current is nil. The Foo1 automaton updates the value of the Current state variable with the thread id (1) of its thread context. In state 2, the model checker selects Foo2 over Bar1, since the value of Current (1) is the same as the thread id of Foo2 and hence Foo2 is chosen over Bar1. Foo2 (and hence Thread1) then blocks in state 3. Bar1 is now chosen to run since there are no other automata that are eligible to run. Bar1 now sets Current to its thread id (2). After this Bar2 runs and then blocks in state 5. Thus we have achieved run-to-completion semantics.

Idle catcher. The combination of maintaining and propagating thread contexts is sufficient as long as there is always an enabled transition in the system. However, there could be problems when there are no enabled transitions in the system, such as when time needs to progress in the model. Figure 3(d) illustrates such a problem, where Thread1 and Thread2 from Figure 3(c) both become unblocked after some time at state 6. At state 6, both Foo2 and Bar2 are enabled and ideally there should be a non-deterministic choice between them. But since the value of Current is 2, Bar2 is always chosen to run by virtue of its thread id being the same as the value of Current. This results in over-constraining the state space, in which a form of nondeterminism which is quite possible and which may be relevant to the constraints of the actual system, is removed. To avoid such over-constraining, we add an Idle_Catcher automaton as Figure 3(d) also shows.

This automaton has a lower preference than any other automaton in the model, and runs only when there are no other enabled transitions in the system. As soon as it runs, the idle catcher automaton resets the currently running thread context to nil. When Foo2 and Bar2 are enabled one of them is picked non-deterministically by the model checker. The selected process then updates the currently running thread context and runs to completion.

5. DOMAIN SPECIFIC OPTIMIZATIONS

A common problem with model checking is the potential for state space explosion. With concurrency the state space can grow especially large due to interleavings of transitions, even when individual processes have relatively small state spaces. Therefore it is imperative to reduce unnecessary nondeterminism and to disable state transitions that do not have a counterpart in the actual system. In this section we describe the techniques we have employed to reduce nondeterminism in the system initialization phase and to limit irrelevant interleavings of state transitions, respectively.

System initialization. When we construct an IF model of a system, we first establish the static structure of the system, creating both active objects (*e.g.*, thread pools) and passive objects (*e.g.*, Reactors) and their associations. In this initialization phase, the order in which the different objects and their associations are created may be irrelevant to the application semantics, in which case they are observationally equivalent. However, different creation orders are by default considered distinct states by the model checker. For example, consider an application with objects A and B that each create an instance of object C. In IF when a process is created (with fork) it gets a unique id. Thus, depending on which object's fork is executed first, we may have the associations $\{A\}0$-$\{C\}0$, $\{B\}0$-$\{C\}1$, or $\{A\}0$-$\{C\}1$, $\{B\}0$-$\{C\}0$. Although these two scenarios are equivalent from an application point of view, they are considered distinct execution paths by the model checker. Since the number of combinations is exponential in the number of such object creations, this can significantly impact the size of the state space. To reduce this type of nondeterminism we arbitrarily choose and fix an object creation order, *e.g.*, in ascending order of process id values, using IF *priority rules*.

Leader thread election. With some concurrency strategies, such as the thread pool reactor, it may not matter in which order a thread is chosen from a set of waiting threads, *e.g.*, to become the *leader thread* and start waiting for events on the reactor. If the choice of a specific thread does not have any consequences for the safety, timing, or liveness properties of the system, then this nondeterminism can be eliminated, thus reducing the state space. We use a simple strategy to remove nondeterminism in this case: among the IF processes representing the waiting threads, we choose the one with the lowest process id number.

Although these techniques are useful in increasing the fidelity of our models and reducing the state space, extreme care is needed in using the above techniques to model real-time middleware. The pre-condition for using each of the above techniques is that the model checker cannot do these optimizations just by using partial order reduction. Wherever necessary, the model checker is provided with extra information necessary for the optimizations (*e.g.*using the priority rules mechanism in IF). However, these technique need to be applied carefully so that subtle forms of interference do not creep in. A formal treatment of these issues is beyond the scope of this paper, but would necessarily involve proof of non-interference and other relevant properties.

6. CASE STUDY: APPLICATION GATEWAY

The case study presented in this section (1) illustrates the reusability of our models and (2) serves as an illustrative example of realistic systems[1], where our low-level models help to identify gaps between higher level models and the actual design and implementation of systems.

This case study illustrates how our low-level models can detect the forms of interference discussed in Section 2, which are not entirely captured by high-level system models like RMA [15]. This case study also shows how our low-level models help to evaluate different middleware-level design alternatives in light of these forms of interference.

Gateway overview. We first give a brief overview of the gateway: a more complete description of this middleware service appears in [23]. The underlying idea of a gateway is the mediator pattern [7] that allows cooperating peers to interact without having to maintain references to each other. A peer that takes the role of a supplier publishes events to the gateway. The gateway forwards these events to peers that take the role of consumers and are subscribed to those events.

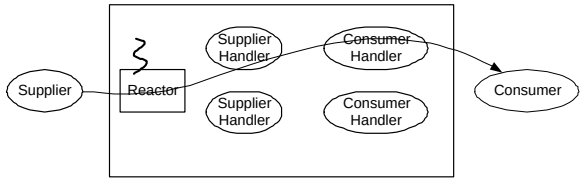
We first extended the default functionality of the gateway so that before forwarding an event to a consumer, the gateway can perform a value-added service that is specific to a supplier and done before forwarding events to each consumer subscribed to that supplier. This is a reasonable extension for real-world applications, *e.g.*, when a stock quote is broadcast to different subscribers the gateway may collect and distribute more information such as the stock's performance history.

[1]Although customers of Riverace (a company that helps to maintain ACE and provides commercial support for a number of ACE-based systems) could not share the details of their use cases with us, Steve Huston, the CEO of Riverace, confirmed that the gateway example is an exemplar of such applications.

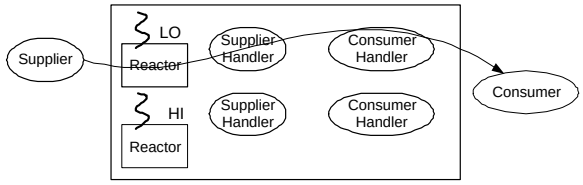
We developed two variations of the gateway, both with event propagation from suppliers to consumers and with suppliers being consumer agnostic. The two variations show how feature additions (*i.e.*, real-time, reliability) can produce changes in the middleware configuration which in turn can affect the timing and liveness properties of the system: (1) a gateway used by an application with real-time requirements; and (2) a gateway used by an application with a control-push data-pull model and reliability requirements.

6.1 Real-time Gateway

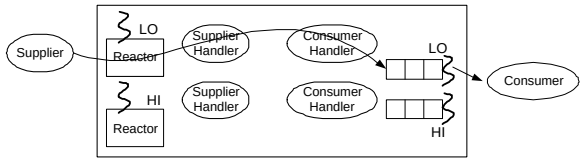
We first examine the use of the gateway by a real-time application. We consider a scenario where events are supplied periodically by two suppliers S1 and S2 with periods 100ms and 50ms respectively. Events from S1 are forwarded to consumers C1 and C2 and events from S2 are forwarded to C2 and C3. Note that C2 receives events from both S1 and S2. The deadlines for the arrival of these events at the consumers is the same as the period of the supplier that supplies the events. The value-added service processing for the events supplied by S1 takes 20ms and that for S2 takes 10ms.



(a) Design 1: Single Threaded



(b) Design 2: Reactor Priority Lanes



(c) Design 3: Reactor and Dispatch Priority Lanes

Figure 4: Real-time Gateway Design Choices

High level modeling using RMA. During high level modeling, we try to determine whether the above application is schedulable under the given parameters. Typically for a periodic system like this, Rate Monotonic Analysis (RMA) [15] is used to determine whether sharing the same CPU among tasks is feasible. Assuming that there is a constant propagation delay from the suppliers to the gateway, the events arrive at the gateway at regular intervals. Under RMA, the gateway thus can be considered a periodic system with 2 periodic tasks (with periods 100ms and 50ms and execution times of 20ms and 10 ms respectively). Since the total utilization (80%) is well below the utilization bound (100%) for harmonic periods, the system is guaranteed to be schedulable if the higher frequency task is given a higher priority, preemptively.

Having done a high-level analysis, we now examine three design choices for configuring the gateway, which are shown in Figure 4. Note that in the RMA analysis, we only considered the sharing of resources at the hardware level and did not consider the sharing of resources that could take place at the middleware level. This lack of detail in the high-level model in turn may lead to a violation of system timing properties during system execution, unless we use a sufficiently detailed model to capture the effects of various design choices thereby guiding the designer to make the appropriate choice. Therefore, we now provide the middleware design details using our own models and analyze the resulting models for any timing violations.

Design 1: single reactor thread. With this design choice, shown in Figure 4(a), an I/O thread waits on socket events using a reactor. When an event arrives from a supplier, the reactor makes an upcall to the appropriate supplier handler which then forwards the event to the appropriate consumer handlers. The consumer handlers send these events to the consumers in the context of the I/O thread itself. Note that the value-added service (if any) for each consumer is also done in the context of the I/O thread.

The model execution trace in Figure 5(a) shows that a deadline miss occurred because of a priority inversion (A) that occurred at the reactor in the gateway. The priority inversion occurred because of the sequential nature of the reactor upcalls. Message from S1 was processed first and then message from S2 was processed. This resulted in a blocking delay for the message from S2. The blocking delay was the time it took for the value-added processing for message from S1, which in the above example was 40 time units (20 time units each for C1 and C2). Because of this blocking delay there was a deadline miss for Consumer C3 at (B). This trace thus shows that the enforcement of the high-level RMA model is not achieved using this design approach.

Design 2: reactor priority lanes. To eliminate the priority inversion due to blocking at the reactor in Design 1, we now use separate reactor/thread pairs to handle I/O events corresponding to the two suppliers. Under this design choice, which has been used to avoid priority inversion in real-time ORBs like TAO [18, 19], there is an I/O thread and reactor per priority level, as is shown in Figure 4(b). The value-added service for each consumer is also done in the context of the I/O thread. To protect the same event handler (for example the consumer handler for C2) from concurrent upcalls from different reactor threads, access to the event handlers is synchronized.

The model execution trace in Figure 5(b) shows that the priority inversion seen at (A) in Figure 5(a)) is prevented because of the priority isolation achieved by separation of I/O handling for the events from the two suppliers. However, a priority inversion still occurs (C) at the synchronized consumer handler corresponding to C2 because the value-added service corresponding to the event from S1 to C2 is done by the synchronized consumer handler for C2. This delays the second event from S2 (released at time = 50) that is waiting for access to the same consumer handler.

Design 3: reactor and dispatch priority lanes. Under this design, a consumer handler hands over an event to an active object [24], which has its own thread of execution to forward the events to a consumer. Synchronization at the event handler is maintained as in Design 2, but the value-added service itself is done by the active object thread rather than within the event handler. To achieve priority isolation for event dispatching by the active object threads, we

used simple Kokyu [8] style priority lanes. The number of lanes is the number of priority levels needed - 2 lanes in this example under RMA, since we have two rate groups (100ms and 50ms).

Our model execution traces indicated no deadline misses or priority inversions, as is shown in Figure 5(c). According to RMA, the lane corresponding to the 100ms period was given a lower priority than that for 50ms. As a result, the S1-C2 event processing by the low priority active object thread is preempted (at time = 50 units) by the S2-C2 event processing by the high priority thread.

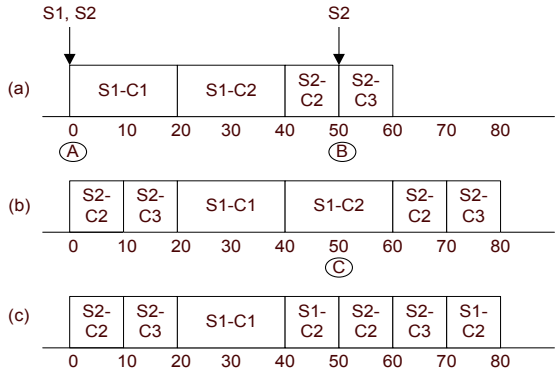


Figure 5: Timelines from Model Execution

6.2 Reliable Gateway (Control-Push Data-Pull)

In Section 6.1 we have seen how our middleware models can help to evaluate different design choices with respect to timing constraints and alternative configurations of the application level gateway. That discussion focused on blocking delays caused by the single threaded reactor or by synchronization at an event handler shared by two different gateway threads.

We now examine a second form of interference that our models capture - exhaustion of reactor threads - using an application with reliability requirements. This example reemphasizes the fact that such interference can be captured only by including lower-level models of middleware building blocks in our analysis. In this example, the application uses an event propagation model called "control-push data-pull" model. In this model, the supplier publishes a "data-available" event to the gateway and the gateway forwards it to the subscribed consumers. The consumers then make remote calls to the supplier to fetch the data.

Apart from the control-push-data pull model, the application also has a reliability requirement - every event that is published by a supplier must be acknowledged by the gateway and every event received by a consumer from the gateway must be acknowledged by that consumer. Once the gateway receives acknowledgements from all the consumers for an event, it sends an acknowledgement back to the supplier.

To wait for an acknowledgement from the gateway after publishing an event, a supplier could use the WaitOnConnection or WaitOnReactor strategy. Which of these strategies is most suitable depends on other factors such as inter-process dependencies and available threads, as well as on application-specific constraints [27]. We now analyze the impact of these two design choices on the liveness of the system, and illustrate how a deadlock may occur in the context of the gateway example as a consequence of using the WaitOnConnection reply wait strategy.

We enhanced both the composed set of low-level models and the implementation of the gateway example in ACE to accommodate reliability. In the following discussion, we consider a single-threaded implementation of the gateway, where a reactor thread is responsible for demultiplexing among connections from suppliers and forwarding the events to consumers. We also assume the suppliers and consumers to have a single-threaded reactor. Scenarios involving multi-threaded suppliers, consumers, and gateways can lead to similar analyses [22] to those presented here, but a detailed discussion of those scenarios is beyond the scope of this paper.

Design 1: reply wait using WaitOnConnection. Figure 6 shows the sequence of interactions drawn from the trace output from our model execution. The trace shows that a supplier first (1) publishes an event to the gateway, and then (2) waits for an acknowledgement from the gateway using the WaitOnConnection (WoC) reply wait strategy. The gateway reactor unblocks, and (3) makes an upcall to the appropriate event handler. The event handler (4) forwards the event to a consumer handler, which then (5) forwards it to the appropriate consumer. The consumer (6) receives the event and makes a remote call to the supplier to get data. After this no transitions are enabled and time advances to a large preset number, indicating a deadlock. The detection of this deadlock exposes an insufficiency of resources (here, reactor threads) for the given construction of the call graph, which can be addressed by enforcing a deadlock avoidance protocol (e.g., BASIC-P or EFFICIENT-P [21]) in the reactors.

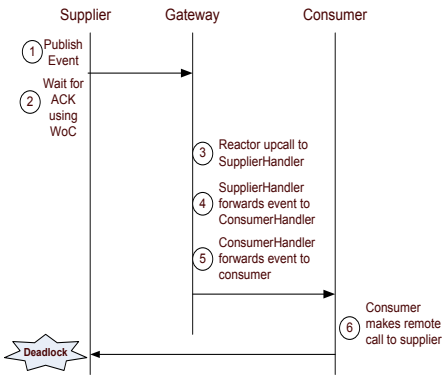


Figure 6: Interaction Sequence: WaitOnConnection

Design 2: reply wait using WaitOnReactor. Figure 7 shows the sequence of interactions drawn from the trace output from our model execution. The trace is similar to the one with the WaitOnConnection strategy (shown in Figure 6) until (6) where a consumer got an event and sends a request to the supplier and waits for a reply. The only difference until (6) in Figure 7 is that the supplier (2) waits for the acknowledgement from the gateway using the WaitOnReactor (WoR) reply wait strategy. After (6), the request sent by the consumer reaches the supplier whose single thread was waiting both for requests and for pending replies, using the reactor. In the WaitOnReactor reply wait strategy, the reactor thread is used to receive the incoming remote call from the consumer and make an upcall to the appropriate event handler. The event handler for the remote call then (7) sends a reply to the consumer, which (8) receives the reply and then (9) sends an acknowledgement to the gateway, which in turn (10) sends an acknowledgement back to the supplier. This trace shows that the WaitOnReactor strategy eliminated the deadlock arising from the loop in the supplier→gateway→consumer→supplier call-chain.

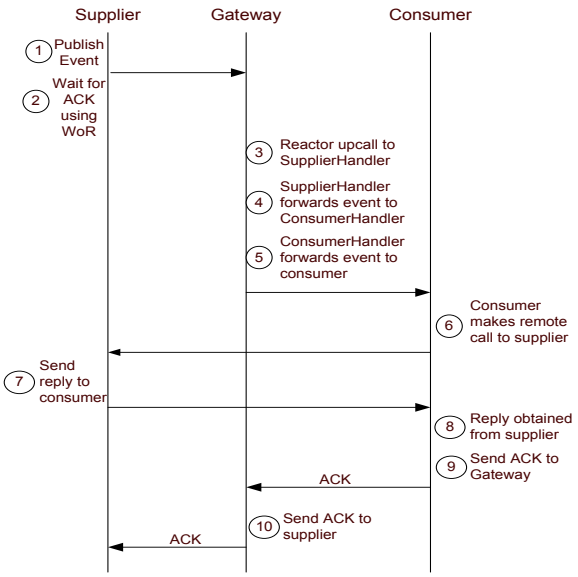


Figure 7: Interaction Sequence: WaitOnReactor

7. RELATED WORK

Model-integrated computing. Our work follows the Model-Driven Middleware [9] paradigm, which applies model integrated computing [28] techniques to the domain of middleware. Our approach provides a detailed set of models for use in conjunction with other model-based middleware configuration techniques such as the CoSMIC [9] tool set, which supports integrated model-driven component assembly, deployment and configuration. We plan to investigate the possibility of integrating our formal models within the Generic Modeling Environment [14] and Ptolemy II [16] environments.

DREAM. DREAM [17, 6] is an open-source tool and method that allows DRE system designers to do model-based schedulability analysis of time and event-driven DRE systems. DREAM offers a computational model called the DRE semantic domain [17]. The key elements in this computational model are tasks, timers, event channels and schedulers. Tasks are triggered either by a timer or external aperiodic events and tasks communicate among themselves by means of an event channel. Within this computational model, DREAM considers the problem of deciding the schedulability of a given set of tasks with time and event-driven interactions. By using timed automata models for each of the elements in the computational model, the schedulability problem is converted [6] into a reachability problem in the composed model using a model checking tool like UPPAAL. DREAM also provides a model transformation facility by which a model of the DRE system expressed using a domain specific modeling language (*e.g.*, ESML [14]), is transformed using model transformation tools to timed automata models in the DRE semantic domain. Even though our approach is similar to DREAM [17, 6] in that we use timed automata models to verify system properties, the problems that these research efforts address are different. Whereas DREAM addresses the problem of deciding schedulability of a set of tasks, our research addresses the problem of correct composition of reusable middleware building blocks that are modeled at a finer level of granularity than the elements in the computational model offered by DREAM.

CADENA and Bogor. CADENA [11] is an integrated environment for building and modeling CORBA Component Model [29] systems. [5] shows how model checking using the extensible Bogor [20] model checker has been applied to verifying event-driven systems using an event channel. We plan to investigate how the low-level formal models we have developed, combined with the middleware building blocks our models represent, could be integrated with these tool sets to provide fine-grained model checking and software synthesis capabilities over a common and reusable software base.

Schedulability Analysis. Model checking provides a common formal basis for checking a wide range of concurrency and timing properties in DRE systems. A variety of analysis techniques for individual properties have been developed in other related work, e.g., for schedulability analysis in tool-sets such as VERSA [4] and Cheddar [25].

8. CONCLUSIONS AND FUTURE WORK

Our middleware modeling approach presented in this paper is designed to address the need for a more detailed formal basis for verification of correct middleware construction and configuration in the context of individual applications. The examples presented in Section 6 illustrate a variety of ways in which evaluating timing and liveness properties can be complicated by different combinations of middleware mechanisms. In practice, the range of complicating factors is much larger than even these examples show, which motivates both our development of reusable mechanism-level models and our composition-based model checking approach for analysis of entire systems. For example, different applications will naturally exhibit (1) different dependency topologies between event handlers; (2) various strategies for concurrency, scheduling, event demultiplexing, and other crucial mechanisms; and (3) alternative strategies for handlers relinquishing control, such as WaitOnConnection and WaitOnReactor. Furthermore, the constraints each application places on timing and other properties may alter the criteria by which system timeliness and liveness are evaluated.

Summary of results. The results of our case study presented in Section 6 motivate the need for detailed modeling of low-level middleware mechanisms, and evaluation of those models through model checking tools. We compared the results of executing our models with the results of executing actual implementations of our case study scenarios with ACE 5.4.7 on Linux 2.6.12, using the design alternatives that we discussed in Section 6. Both the the priority inversions predicted by the models in Section 6.1, and the deadlocks predicted by the models in Section 6.2, appeared in the actual runs, thus demonstrating the validity of our models.

Moreover, for the real-time gateway scenarios in Section 6.1, we populated the models with execution times from the actual runs and then generated timeline traces from the resulting model execution. The timelines from the model execution trace resembled the timelines from actual execution trace very closely, demonstrating the fidelity of our models. A more detailed discussion of our modeling approach, of this case study, and of other verification and validation examples using our models can be found in [26].

These results support the view that modeling and analysis should be done as an integral part of the system design and engineering process. Significant further work is needed to make this vision a reality in the DRE middleware domain, but the research presented in this paper demonstrates the suitability and viability of that approach.

Acknowledgments

We wish to thank Dr. Joseph Sifakis, Dr. Marius Bozga and Dr. Iulian Ober for valuable discussions regarding the IF tool set.

9. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *COMPOS*, pages 103–129. Springer-Verlag LNCS 1536, 1997.

[3] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF Toolset. In *Formal Methods for the Design of Real-time Systems*. Springer-Verlag LNCS 3185, 2004.

[4] D. Clarke, I. Lee, and H.-L. Xie. Versa: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3(2), apr 1995.

[5] W. Deng, M. B. Dwyer, J. Hatcliff, G. Jung, Robby, and G. Singh. Model-checking Middleware-based Event-driven Real-time Embedded Software. Department of Computer Science, Technical Report SAnToS-TR2003-2, Department of Computing and Information Sciences, Kansas State University, 2003.

[6] Gabor Madl and Sherif Abdelwahed and Gabor Karsai. Automatic Verification of Component-Based Real-time CORBA Applications. In *The 25th IEEE Real-time Systems Symposium (RTSS'04)*, Lisbon, Portugal, Dec. 2004.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] C. Gill, D. C. Schmidt, and R. Cytron. Multi-Paradigm Scheduling for Distributed Real-time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), Jan. 2003.

[9] A. Gokhale, B. Natarajan, D. C. Schmidt, A. Nechypurenko, J. Gray, N. Wang, S. Neema, T. Bapty, and J. Parsons. CoSMIC: An MDA Generative Tool for Distributed Real-time and Embdedded Component Middleware and Applications. In *Proceedings of the OOPSLA 2002 Workshop on Generative Techniques in the Context of Model Driven Architecture*, Seattle, WA, Nov. 2002. ACM.

[10] S. Graf, I. Ober, and I. Ober. Model-checking UML models via a mapping to communicating extended timed automata. In *Proceedings of SPIN'04*, 2004.

[11] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, May 2003.

[12] Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University.

[13] Institute for Software Integrated Systems. The ADAPTIVE Communication Environment (ACE). www.dre.vanderbilt.edu/ACE/, Vanderbilt University.

[14] G. Karsai, S. Neema, A. Bakay, A. Ledeczi, F. Shi, and A. Gokhale. A Model-based Front-end to ACE/TAO: The Embedded System Modeling Language. In *Proceedings of the Second Annual TAO Workshop*, Arlington, VA, July 2002.

[15] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-time Environment. *JACM*, 20(1):46–61, Jan. 1973.

[16] J. Liu, X. Liu, and E. A. Lee. Modeling Distributed Hybrid Systems in Ptolemy II. In *Proceedings of the American Control Conference*, June 2001.

[17] G. Madl and S. Abdelwahed. Model-based analysis of distributed real-time embedded system composition. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 371–374, New York, NY, USA, 2005. ACM Press.

[18] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. Levine. Evaluating Policies and Mechanisms for Supporting Embedded, Real-time Applications with CORBA 3.0. In *Proceedings of the* 6^{th} *IEEE Real-time Technology and Applications Symposium*, Washington DC, May 2000. IEEE.

[19] I. Pyarali, D. C. Schmidt, and R. Cytron. Techniques for Enhancing Real-time CORBA Quality of Service. *IEEE Proceedings Special Issue on Real-time Systems*, 91(7), July 2003.

[20] Robby and Matthew Dwyer and John Hatcliff. Bogor: An Extensible and Highly-Modular Model Checking Framework. In *In the Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, Sept. 2003. ACM.

[21] C. Sanchez, H. B. Sipma, Z. Manna, V. Subramonian, and C. Gill. On Efficient Distributed Deadlock Avoidance for Real-time and Embedded Systems. In *20*th *IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, Apr. 2006.

[22] C. Sanchez, H. B. Sipma, V. Subramonian, C. Gill, and Z. Manna. Thread Allocation Protocols for Distributed Real-time and Embedded Systems. In *25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '05)*, oct 2005.

[23] D. C. Schmidt. Applying a Pattern Language to Develop Application-level Gateways. In L. Rising, editor, *Design Patterns in Communications*. Cambridge University Press, 2000.

[24] 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.

[25] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar: a flexible real time scheduling framework. *Ada Letters*, XXIV(4):1–8, 2004.

[26] V. Subramonian. *Timed Automata Models for Principled Composition of Middleware*. PhD thesis, Washington University in St. Louis, Computer Science and Engineering Department Technical Report WUCSE-2006-23, May 2006.

[27] V. Subramonian and C. Gill. A Generative Programming Framework for Adaptive Middleware. In *Proceedings of the 37th Hawaii International Conference on Computer Sciences (HICCS)*, Kona, Hawaii, Jan. 2004.

[28] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, Apr. 1997.

[29] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian. Configuring Real-time Aspects in Component Middleware. In *Lecture Notes in Computer Science: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04)*, volume 3291, pages 1520–1537, Agia Napa, Cyprus, Oct. 2004. Springer-Verlag.