Report Number: WUCSE-2005-54

2005-11-23

Composable Models for Timing and Liveness Analysis in Distributed Real-Time Embedded Systems Middleware

Venika Subramonian, Christopher Gill, Cesar Sanchez, and Henry B. Sipma

Middleware for distributed real-time embedded (DRE) systems has grown increasingly complex, to address functional and temporal requirements of diverse applications. While current approaches to modeling middleware have eased the task of assembling, deploying and configuring middleware and the applications that use it, a lower-level set of formal models is needed to uncover subtle timing and liveness hazards introduced by interference between and within distributed computations, particularly in the face of alternative middleware concurrency strategies. In this paper, we propose timed automata as a formal model of low-level middleware building blocks from which a variety different middleware configurations can be... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

[Department of Computer Science & Engineering](#) - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Composable Models for Timing and Liveness Analysis in Distributed Real-Time Embedded Systems Middleware

Venika Subramonian, Christopher Gill, Cesar Sanchez, and Henry B. Sipma

Complete Abstract:

Middleware for distributed real-time embedded (DRE) systems has grown increasingly complex, to address functional and temporal requirements of diverse applications. While current approaches to modeling middleware have eased the task of assembling, deploying and configuring middleware and the applications that use it, a lower-level set of formal models is needed to uncover subtle timing and liveness hazards introduced by interference between and within distributed computations, particularly in the face of alternative middleware concurrency strategies. In this paper, we propose timed automata as a formal model of low-level middleware building blocks from which a variety different middleware configurations can be constructed. When combined with analysis techniques such as model checking, this formal model can help developers in verifying the correctness of various middleware configurations with respect to the timing and liveness constraints of each particular application.

Composable Models for Timing and Liveness Analysis

in Distributed Real-Time Embedded Systems Middleware

Venkita Subramonian and Christopher Gill*   César Sánchez and Henny B. Sipma†

{venkita,cdgill}@cse.wustl.edu   {cesar,sipma}@cs.stanford.edu

CSE Department, Washington University, St. Louis, MO   CS Department, Stanford University, Stanford, CA

Abstract

Middleware for distributed real-time embedded (DRE) systems has grown increasingly complex, to address functional and temporal requirements of diverse applications. While current approaches to modeling middleware have eased the task of assembling, deploying and configuring middleware and the applications that use it, a lower-level set of formal models is needed to uncover subtle timing and liveness hazards introduced by interference between and within distributed computations, particularly in the face of alternative middleware concurrency strategies. In this paper, we propose timed automata as a formal model of low-level middleware building blocks from which a variety different middleware configurations can be constructed. When combined with analysis techniques such as model checking, this formal model can help developers in verifying the correctness of various middleware configurations with respect to the timing and liveness constraints of each particular application.

Keywords. Model-driven Middleware, Embedded Systems Middleware, Formal Middleware.

Technical Area. Operating Systems and Middleware

1 Introduction

Significant research over the past decade has made middleware more customizable through the use of pattern-oriented software frameworks [1, 2]. Although this has made middleware solutions suitable for a wider range of applications, managing the resulting multiplicity of customization options has become an increasing concern. To allow middleware to be customized to meet the stringent demands of different distributed real-time embedded (DRE) applications, recent research has focused on applying model-driven techniques to DRE middleware [3]. Although current model-driven middleware approaches *facilitate* the correct assembly, deployment and configuration of DRE applications and middleware, we argue in this paper that a more detailed and formal basis for reasoning about timing and liveness properties in a variety of different middleware configurations is both necessary and possible.

Formal models have been used to uncover high-level design flaws early in system development [4, 5]. However, such models are currently difficult to maintain adequately as the system's specification is refined successively throughout the system development

life-cycle. For example, decisions regarding the deployment of application components onto end-systems, or the choice of middleware concurrency strategies, often are not reflected in these high-level models. This may result in subtle timing and liveness hazards from unexpected *interference* between the middleware policies and mechanisms used by a set of distributed computations.
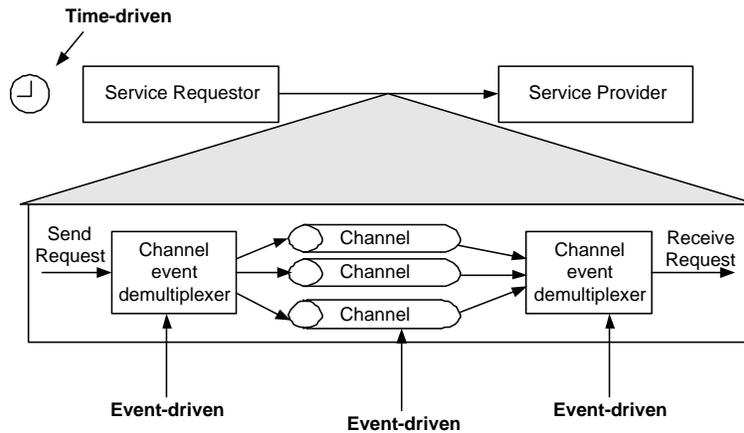
Figure 1. Remote Function Call as a Time and Event-driven Interaction

Figure 1 shows how a system that is specified with only time-driven constraints in its high-level model may be refined into a time and event-driven system during its design and implementation phases. In the high-level model, a purely time-triggered request is sent from a service requester to a service provider through a middleware-implemented remote function call. However, the implementation of this remote function call goes through multiple middleware, OS and network processing stacks, each of which likely contains event-driven system elements. For example, Figure 1 shows middleware based event demultiplexers on the sender and receiver side, which enable a single thread on each side to be used to demultiplex I/O events (*e.g.*, packet arrivals and transmissions) from and onto multiple interaction channels (*e.g.*, sockets and pipes). Even the interaction channels are likely to be event driven, for example when IP packets arrive and are moved from the network interface card into an application-accessible transport-layer buffer.

This example illustrates the general concern that many of the abstractions used during high-level modeling, such as the notion of a purely time-driven interaction between the service requester and the service provider, may become decreasingly representative of the system during its design and implementation. This may in turn result in a chasm between the high-level model and the actual implementation, *unless the abstractions used in the high level model can be refined during design and implementation*. Thus, a foundational set of formal models that can express both (1) high-level abstractions such as timed remote method invocations, and (2) low-level refinements such as concurrency and interaction semantics between the objects that implement the high-level model, is needed to support verification of the high level model in terms of its low-level design and implementation.

Furthermore, the insights obtained from modeling and analysis should be made available and used while making design and development decisions, and vice-versa. Such a close correspondence between the system modeling, analysis, design and development activities offers the following benefits: (1) more complete, detailed and executable models of systems, including their middleware infrastructure, can be composed and checked; (2) timing and liveness properties can be verified with greater precision; (3) a more rigorous and formal style of documentation can be used to capture and communicate detailed middleware engineering expertise that is currently represented less formally, *e.g.*, as *patterns* [6]; (4) with more representative models and more powerful verification techniques, the extent to which systems must be "over-designed" can be reduced due to greater insight into the possible behaviors of the system.

In this research, we are developing timed automata [7] models of canonical lower-level middleware building blocks that have been used to implement a wide range of middleware frameworks. These timed automata models can then be combined with higher-level formal models to provide a faithful model of a system *including the middleware platform on which the system is deployed*, such that the composite models can be verified for correctness with higher fidelity to the system itself. Performing such verification at a realistic scale will require significant work. We propose an approach that combines the use of protocols that are provably correct with respect to certain properties, with the use of model checking and deduction to verify the composition of these protocols with other middleware building blocks and application layer programs. In [8, 9] we presented such provably correct protocols for deadlock avoidance in systems with nested upcalls. In this paper we focus on creating models for combinations of middleware mechanisms and their exploration with the model exploration tool IF [10]. Our long term objective is to add rigor to the model-based approaches to middleware development currently being pursued by the systems research community, and to provide high-fidelity composable models of foundational middleware building blocks to the formal methods community.

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 our modeling approach and the middleware architecture that is captured by our models. Section 4 discusses challenges and solution approaches for tractable checking of the models we have developed. In Section 5 we present a case study in which we use our models to analyze the impact of a specific deadlock avoidance protocol that we developed in previous work [8, 9]. In Section 6, we present a case study of scenarios in which *interference* can be caused by a broader set of middleware concurrency and interaction strategies, which in turn affect system timing and liveness properties. Section 7 examines the results of model checking and empirical evaluation of the case studies presented in Sections 5 and 6. Section 8 describes related work, and Section 9 offers concluding remarks and summarizes future work.

2   System Model and Problem Definition

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 end-system *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 by a natural number, 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 events in our system model.

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:

- $\alpha_H : H \to 2^A$. The set of actions that can be taken on event handler h is given by $\alpha_H(h)$.
- $\alpha_I : I \to 2^A$. The set of actions that can be taken on interaction channel i is given by $\alpha_I(i)$.
- $\alpha_R : R \to 2^A$. The set of actions that can be taken on reactor r is given by $\alpha_R(r)$.
- $threadpool : R \to 2^\theta$. 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.

Further subdivision of the thread pools may be useful for some kinds of middleware, for example to model thread pools at different priorities within a reactor [11], but for the scenarios considered in this paper, we can assume a single thread pool per reactor. It is also possible for an end-system to employ additional threads that are not assigned to a reactor, but a useful middleware design idiom is only to use threads from reactor thread pools so that it is easier to apply policies such as prioritization consistently across all threads. Therefore, we confine our attention in this paper to the case where all end-system threads are in reactor thread pools.

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:

- $registered : E \times I \times R \times T \to 2^H$. The set of event handlers registered for event e on interaction channel i in reactor r at time t is given by $registered(e, i, r, t)$.
- $active : R \times T \to 2^E$. The set of events that have arrived at reactor r but have not been dispatched to event handlers at time t is given by $active(r, t)$.
- $ready : E \times R \times T \to 2^I$. 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 one read from a socket for a "data ready" event, can be taken on a ready interaction channel without blocking the thread in which that action is taken.
- $dispatched : R \times T \to 2^\theta$. The set of threads in $threadpool(r)$ that are in use at time t dispatching 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 $dispatched(r, t)$.
- $blocked : R \times T \to 2^\theta$. 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 performing a blocking read on a socket, an event to trigger unblocking must be generated by an action taken by another thread.
- $deadline : \mathcal{N} \times E \to T$. The time by which the nth occurrence of event e is constrained to occur is given by $deadline(n, e)$. Event occurrences that do not have timing constraints are assumed to have a deadline of ∞.
- $occurred : \mathcal{N} \times E \to T$. The time at which the nth occurrence of event e happened is given by $occurred(n, e)$.
- $live : R \times T \to 2^\theta$. The set of threads assigned to reactor r within each of which at least one action occurs after time t is given by $live(r, t)$.

Problem definition. Our approach hinges on the idea that *interference* occurs when the actions taken by end-system 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 end-system middleware can cause violations of application-specific timing and liveness constraints.

To detect this kind of interference, we use model checking to 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 *deadlock* which is a liveness constraint violation that usually also leads to timing constraint violations in subsequent system states. Checking for a missed deadline in a state can be done using our system model by comparing the time at which an occurrence n of an event e happened, to the deadline for that occurrence of the event: $occurs(n, e) > deadline(n, e)$. Deadlocks can be detected in 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 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 can then 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 new techniques that can in some cases prevent, or in others at least help avoid, those errors. In Sections 3 and 4 we describe the models we have developed in this research, and our use of the modeling and model checking tools within which we represent and explore them. In Sections 5 and 6 we present case studies of how different forms of interference can arise, and how model checking can be used to detect them or to verify their absence.

3 Modeling Approach

To be able to verify the correctness of customized middleware 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 [1] framework, such as reactors, thread pools, event handlers, and interaction channels, which can be composed and checked rigorously to evaluate timing and liveness properties in each particular application and its supporting middleware configuration. A crucial challenge is to determine the appropriate level of abstraction at which to model system software. To answer this question, one must look at the kinds of abstractions used in state-of-the art system implementations. For example, distribution middleware services such as CORBA [12] object request brokers (ORBs) provide a level of abstraction that promotes portability and reusability and hence makes an appealing candidate for formal modeling. Since many state-of-the-art distribution middleware implementations expose sets of configuration options used to tailor the middleware to particular applications, modeling the combinations of configuration options [3] is a useful and necessary step toward model-driven construction and verification of DRE systems. We contend, however, that to evaluate issues such as timing and liveness, which are crucial to many DRE systems, finer-grained models of lower-level middleware building blocks are needed to capture and supplement analysis of crucial details related to concurrency and interaction.

The problem definition given in Section 2 guides our selection of models for analysis of timing and liveness. We rely first on

model checking to ensure soundness. Due to the potential size of the state spaces that need to be checked, we then apply several optimizations: (1) building highly modular models, by sub-dividing them into fine-grain composable automata; (2) encoding our models in formats used by model checkers that allow automata to be added to a model, or removed from it, dynamically; and (3) adopting a hybrid approach in which parts of the analysis are provided by other analysis techniques [8, 9] thus reducing the state space that must be explored through model checking. Model checkers such as UPPAAL [13], IF [10], Bogor [14], and SPIN [15] each have their particular features and restrictions. For example, among these four tools, timed automata models are supported only by UPPAAL and IF, whereas only Bogor supports object-oriented concurrent constructs explicitly. UPPAAL uses a rendezvous model of communication whereas in IF communication is asynchronous. Because our middleware models must capture time, concurrency, and asynchronous interactions between system elements that can be added and removed dynamically, we have selected IF as the most suitable model checking environment for our purposes.

3.1 Middleware Modeling Architecture

Figure 2 shows our modeling architecture, which we have implemented for the IF tool set [10, 16, 17]. We use the IF (Intermediate Format) notation to specify our fine-grained models as *processes* 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* [18] 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: models of network and OS level abstractions such as channels for interprocess communication; models of semantically rich middleware building blocks like reactors; and models of the application functionality implemented in the form of event handlers. Although Figure 2 shows a static view of our models, 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
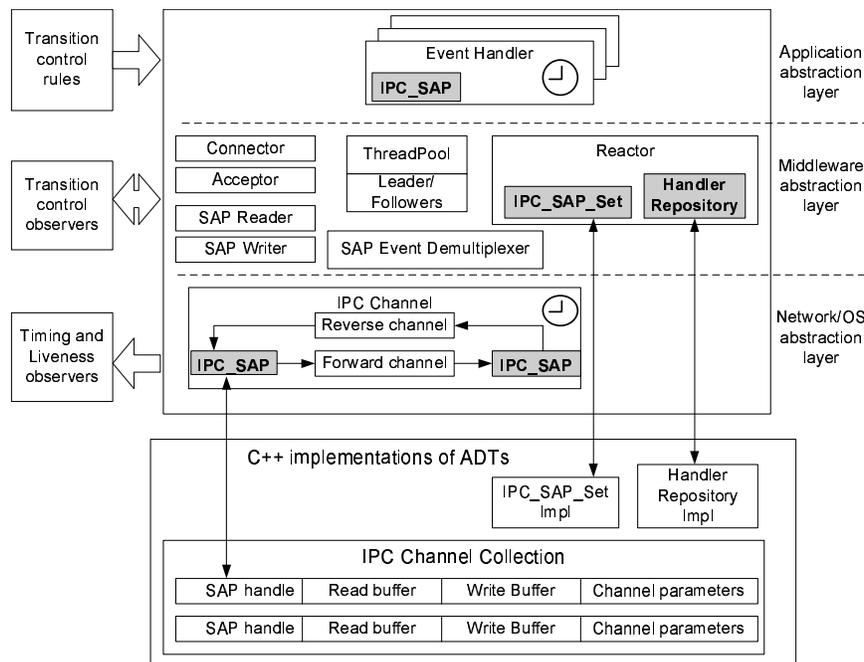
Figure 2. Middleware Modeling Architecture

are modeled using timed finite state automata specified using the IF language. The shaded rectangular boxes shown in Figure 2 are

abstract data types that are used to integrate the models, but their implementations are kept outside of the models to reduce the state space and have only a minimal representation in the models, through specific interfaces to the data. Using these interfaces, the data outside the model can be manipulated using functions written in C++. Timed transitions (transitions that are guarded with conditions based on clock variables) are indicated in Figure 2 by timer icons. We now describe each of the layers of our middleware modeling architecture in greater detail.

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 (SAP), 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. These buffers are not exposed in the model, since the data itself does not play any significant role in the kind of properties, *i.e.*, timeliness and liveness requirements, with which our research is concerned. Instead, the read and write buffers associated with each SAP are stored as part of an IPC channel collection outside of the model. Each SAP has a unique handle associated with it and this handle is used internally in the C++ IPC channel collection data structure to access the data buffers associated with a SAP.

(A)
```
type IPC_SAP = record
    sap_handle integer;
    suspended integer;
    annotation integer;
endrecord;
```

(B)
```
type IPCC_Collection =
    abstract
        integer is_any_sap_hot(IPCC_Collection,
                IPC_SAP_Set, IPC_SAP_Set);
    endabstract;
```

(C)
```
procedure IPCCC_enqueue_data;
fpar inout ipccc IPCC_Collection,
    in sap IPC_SAP, in qtype IPCQType,
    in num_bytes integer;
returns integer;
{#
  int rc;
  rc = ipccc.enqueue_data(sap, qtype,
                    num_bytes);
  return rc;
#}
endprocedure;
```

(D)
```
procedure IPCCC_get_data;
fpar inout ipccc IPCC_Collection,
    in sap IPC_SAP, in qtype IPCQType,
    in num_bytes integer;
returns integer;
{#
  int rc;
  rc = ipccc.get_data(sap, qtype, num_bytes);
  return rc;
#}
endprocedure;
```

(E)
```
struct IPC_Queues
{
    int readq;
    int writeq;
};
typedef std::map<int,IPC_Queues> IPC_Queues_Map;

class IPCC_Collection
{
    IPC_Queues_Map ipc_queues_map_;
    int next_avail_sap_handle_;
```

(F)
```
int get_data(const if_IPC_SAP_type& sap,
            if_IPCQType_type qtype, int num_bytes)
{
  IPC_Queues_Map::iterator ipcq_iter =
        ipc_queues_map_.find(sap.sap_handle);
  if (qtype == if_IPCQ_READ_constant)
    ipc_queues_map_[sap.sap_handle].readq -= num_bytes;
  else if (qtype == if_IPCQ_WRITE_constant)
    ipc_queues_map_[sap.sap_handle].writeq -= num_bytes;
  return num_bytes;
}
```

(G)
```
int enqueue_data(const if_IPC_SAP_type& sap,
            if_IPCQType_type qtype, int num_bytes)
{
  IPC_Queues_Map::iterator ipcq_iter =
        ipc_queues_map_.find(sap.sap_handle);
  if (qtype == if_IPCQ_READ_constant)
    ipc_queues_map_[sap.sap_handle].readq += num_bytes;
  else if (qtype == if_IPCQ_WRITE_constant)
    ipc_queues_map_[sap.sap_handle].writeq += num_bytes;
  return num_bytes;
}
```

Model inside IF   C++ data structure outside IF

Figure 3. IPC Channel Collection Model and Data Structure Extracts

Figure 3 shows extracts from the the IPC channel collection model and data structure. IPC_SAP is declared as an IF *record* (A) and the IPC channel collection is declared as an *abstract data type* (ADT) (B) in the IF model. Two IF *procedures* (C and D) are used to encapsulate C++ functions (F and G), which in turn access the IPC channel collection data structure. In this case, the IPC channel collection data structure is implemented as a C++ Standard Template Library *map* (E). It should be noted that we model the read and

write buffers (E) that are associated with an IPC_SAP as integers. Since the properties that we are analyzing are not influenced by the actual *contents* of the read and write buffers, it is sufficient to store only the number of bytes contained in the read and write buffers [1], which helps in reducing the state space of the models. Figure 3 also illustrates calls to the functions to get (D) data from and put (C) data into the read and write buffers associated with a SAP. These can be called from inside the model to access and enqueue data for a SAP. The IPCQ_Type parameter (F and G) specifies the type of buffer (read or write). In the C++ IPC channel data structure, we access the read and write buffers for a SAP in the map using the SAP's unique handle and then increment the write buffer counter by the number of bytes to be written during a write operation (G) or decrement the read buffer counter by the number of bytes to be read during a read operation (F). Parameterizing the number of bytes to be read or written makes it easy to model the read and write OS system calls respectively. An example of this kind of usage can be seen in the IPC channel data structure that we describe in detail next.

An IPC channel is bidirectional. It is modeled, however, 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 propagation delays, if needed.

Middleware abstraction layer. The next architectural layer above the network/OS layer is the middleware layer, where we model abstractions of semantically rich middleware building blocks. Here we use ADTs to encapsulate 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.

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. For example, the reactor IF model uses the IPC channel collection ADT methods to query the I/O status of different SAPs - *e.g.*, whether data is ready to be read or written. Using the provided IF clause, it waits for I/O events on a set of SAPs. Once the event arrives its guard condition becomes true, and it then uses IF procedures that are defined in the model to obtain the read-ready and write-ready SAPs. For each of the *ready* SAPs, the handle repository is accessed using another procedure in our model to obtain the corresponding event handler. The handle_input or handle_output IF signal is then sent to the event handler, depending on whether the SAP is ready for reading or writing. Sending these signals models the initiation of the respective handle_input and handle_output upcall dispatch actions performed by the actual reactor implementation.

Application abstraction layer. In our models, we abstract the application functionality using event handlers as is customary when developing ACE applications in practice [19, 20]. 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 is abstracted away and represented by a single transition, guarded by a constraint on a timer variable to delay its execution as necessary. An event handler reads and writes data to an SAP using the IF procedures described before.

---

[1]We count the number of bytes in our models so that as future work we can consider bandwidth and other spatio-temporal features - for the concerns addressed in this paper, however, a Boolean value would have been a sufficient representation.

Property specifications for verification. In the IF toolset properties can be specified by observers [10]. Like the system model, 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. The observer can also be used to control the extent of the state space that is explored using the cut statement, which cuts off selected execution paths. An observer can also be intrusive and act as an *interceptor* [6] and change the system state by changing variables or sending signals. Examples of such observers are shown in Section 4, where we use an intrusive observer to help reduce the state space.

3.2 Issues for Modeling Concurrent Object-Oriented Systems in IF

The IF toolset was developed for the analysis of communicating processes. Its basic construct for representing behavior is a *process*. IF does not directly support the notions of object or thread. Thus it is the responsibility of the IF model developer to distinguish between these concepts in the model. In particular, in concurrent systems, 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.

To enable dynamic creation of processes, the IF language provides the fork construct: as part of a state transition, the fork action creates a new instance of a named process. Similarly, processes can be deleted with stop. A method call from one object to another object can now be modeled by the caller process creating a new process for the callee method and sending a signal to the new process to start its execution [21]. Upon completion of execution the callee process sends a signal back to the parent (caller) process and stops, thus deleting itself. Despite its lack of direct support for objects and threads we decided to use IF as our tool set for the analysis of DRE systems (rather than, for example, Bogor [14], which provides native support for objects and threads) because of IF's ability to support reasoning about real-time properties, which Bogor does not provide.

4 Domain Specific State Space Optimizations

Our objective is to apply model checking to verify temporal properties of models of DRE systems. A common problem with model checking is the state space explosion problem. Especially in the presence of concurrency the state space can grow very large due to the many interleavings, even with individual processes with relatively small state spaces. Therefore it is imperative to minimize nondeterminism and disable state transitions that do not have a counterpart in the actual system. Below we describe the techniques we have employed to reduce nondeterminism at the system initialization phase and to limit interleavings at the execution state, respectively.

4.1 Ordering Optimizations

Because our models allow different interleavings of actions, particularly when the models represent multiple threads of execution, it is important to distinguish interleavings of actions that are relevant to the application constraints, from spurious interleavings that could

easily render the model's state space intractable. We first examine interleavings caused by the order in which objects are initialized or the order in which threads waiting for a synchronization token are chosen.

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. ThreadPools, and passive objects, e.g. Reactors, and their associations. We use the IF process construct to model both kinds of objects, so a Reactor, even though it is a passive object, is modeled by an automaton and can run concurrently with the other automata. At the initialization phase the order in which the different objects and their associations are created is irrelevant to the application semantics: they are observationally equivalent. However, different orders are, by default, considered distinct states by the model checker. For example, consider an application with objects A and B that each create (fork) 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 $\{A\}0\text{-}\{C\}0$, $\{B\}0\text{-}\{C\}1$, or $\{A\}0\text{-}\{C\}1$, $\{B\}0\text{-}\{C\}0$. Although these two scenarios are equivalent from an application point of view, they are considered distinct states 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 eliminate this type of nondeterminism we arbitrarily choose and fix an object creation order, e.g. ascending order of process id values, using the IF *priority rules*.

Leader thread election. With some concurrency strategies, such as the thread pool reactor described in Section 5.1, it may not matter in which order a thread is chosen from a set of waiting threads, *e.g.*, to become the leader thread to 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 non-determinism can be eliminated, thus reducing the state space. We use a simple strategy to remove non-determinism in this case: among the IF processes representing the waiting threads, we choose the one with the lowest process id number.

4.2 Run-to-Completion Semantics

In real-time operating systems, it is very common to use the SCHED_FIFO scheduling mechanism to control preemption between real-time threads of the same priority. We use a similar technique in our models to control interleaving between two threads. Since IF does not have distinct native support for constructs like threads and objects, 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. To achieve this, we use the concept of a *thread id* maintained by each IF process. This serves to record the real-world thread context under which that IF process is executing. When we model an object method invocation, the thread context under which that invocation is made must be carried over from the caller to the callee object. For example, Figure 4 shows a logical thread that represents a flow of control from one object (P) to another object (Q), with both of these objects modeled as IF processes.
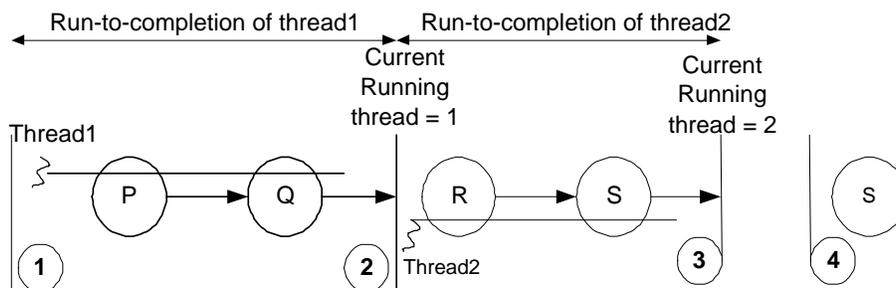

Figure 4. Run-to-Completion Semantics for Two Threads

10

We model SCHED_FIFO semantics to control unnecessary interleavings. Each logical thread of control will run across multiple IF processes until the thread blocks, and only then can another thread of control start running. Figure 4 illustrates this situation for two threads of control. Each of these threads passes through multiple objects. Thread1 flows through objects P and Q and Thread2 flows through objects R and S. Thread1 runs to completion before Thread2 can run, where completion means that a thread completes a phase of its activity that is totally CPU bound.

In the IF model, this translates to the notion of processes in the same thread context executing in sequence until there are no enabled transitions in the group of processes running under that thread context. To realize the run-to-completion semantics in IF, we used 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.

Currently running thread context. Each transition in every process in the model updates this state variable to record the thread context under which it is currently running. Any IF process whose thread context is the same as the currently running thread will get preference to any IF process whose thread context is not the same as the currently running thread. This policy can be expressed in IF using a combination of IF priority rules.

Thread context propagation. To realize the run-to-completion semantics in IF, it is not sufficient that a globally accessible state variable is updated with the current thread context. This is because the priority rules in IF are executed in the context of the current global state of the system. The state of a process must be updated with the thread context under which it is running, *before* the execution of priority rules, since this is the state that is used by the priority rules. With the first technique above, this update happens *after* a decision is made in the model checker as to which process to execute next from among the list of processes with enabled transitions. The first step is necessary, however, to allow non-determinism in the system, and consequently whichever process runs first runs to completion. To propagate the thread context, we use an IF intrusive observer. Between any two *labeled transition system* (LTS) steps, this 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. The observer observes the output of a signal from a source process to a destination process and updates the thread context of the destination process to be the same as the source process. This is done *before* the execution of priority rules and since the thread context of processes have already been established, only one process will run. This method has the disadvantage that if a process sends multiple signals to other processes, then it will result in some non-determinism in choosing between the destination processes. In our case, however, we use this technique for modeling thread context propagation along object method invocations, where one method invocation has only one destination.

Idle catcher. The combination of the two previous techniques 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, for example, when time needs to progress in the model. Figure 4 illustrates such a problem, where Thread1 (at process P) and Thread2 (at process R) are enabled at (1), where a non-deterministic choice is made between P and R. Assuming that process P is selected to run by the model checker, Thread1 blocks when process Q blocks at (2) waiting for some event. Process R is then selected to run and Thread2 runs to completion at (3). Note that the current running thread is updated at (1) and (2) to be Thread1 and Thread2 respectively. At (3) Thread2 blocks,

when S blocks waiting on some event. At (3), the current thread is still recorded as being Thread2. As a consequence, at (4) when Q and S are both enabled, only S is selected by the model checker since the current thread is recorded as being Thread2. This results in over-constraining the state space, in which a form of non-determinism which is quite possible and which may be relevant to the constraints of the actual system that we are trying to model, is removed. To avoid such over-constraining, we add an Idle_Catcher process as Figure 5 shows. This process has a lower preference than any other process in the model, and runs only when there are
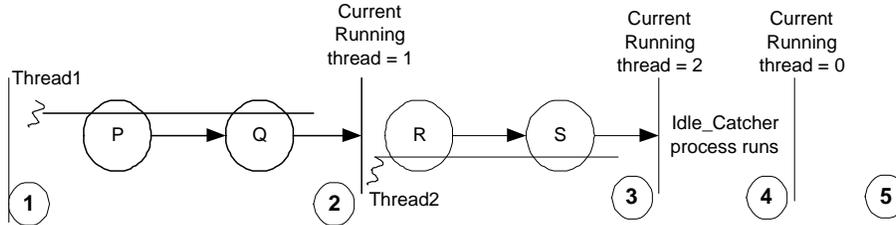
Figure 5. Run-to-Completion Semantics with Idle Catcher

no other enabled transitions in the system (3). As soon as it runs, it resets the state variable that stores the currently running thread (4). Now, when Q and S are enabled (5) 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   Case Study: Deadlock Avoidance Protocol

In complementary research [8], we have developed thread allocation protocols for deadlock avoidance that exploit information about the application's call graph, *e.g.*, the depth of nesting at each position of each call chain. The deadlock avoidance protocol is a hybrid technique that uses both static call graph analysis and run-time protocol code to avoid deadlocks. The outcome of the static call graph analysis is a set of integer *annotations* to each node in the call graph. The annotations can be chosen using different algorithms [8, 9], which balance efficiency and correctness according to application-specific criteria. Once the annotations are assigned, the run-time implementation of the protocol is used to grant or delay dispatching of events within a reactor, to avoid deadlock.

5.1   Implementation of Deadlock Avoidance Protocols in ACE ThreadPool Reactor

This section discusses our implementation of the deadlock avoidance protocol discussed in [8] in the context of the ThreadPool (TP) reactor [6] in ACE [1]. The ACE TP reactor uses the Leader-Followers [6] pattern to share the same reactor instance among multiple threads in a thread pool. The Leader-Followers pattern has several benefits [6]: (1) it reduces the number of context switches when delivering upcalls since the I/O operation takes place in the same thread context as the event handler upcall; (2) it increases throughput by sharing the workload among multiple worker threads; and (3) it supports long-running service handlers by allowing each such handler to run in the context of one thread in the thread pool while another thread from the same pool waits on the reactor to demultiplex and dispatch other concurrent I/O events.

Figure 6 shows how we have implemented support for deadlock avoidance (DA) protocols in the context of the ACE TP Reactor without incurring meaningful overhead (as we show in Section 7) for use cases that do not use a DA protocol. The additional components that we have introduced to the existing reactor framework in ACE to support DA protocols are shown with a shaded background in Figure 6. We now summarize the sequence of events and actions that occur in the TP reactor, and indicate where we

12

have added deadlock avoidance protocol support in this context:
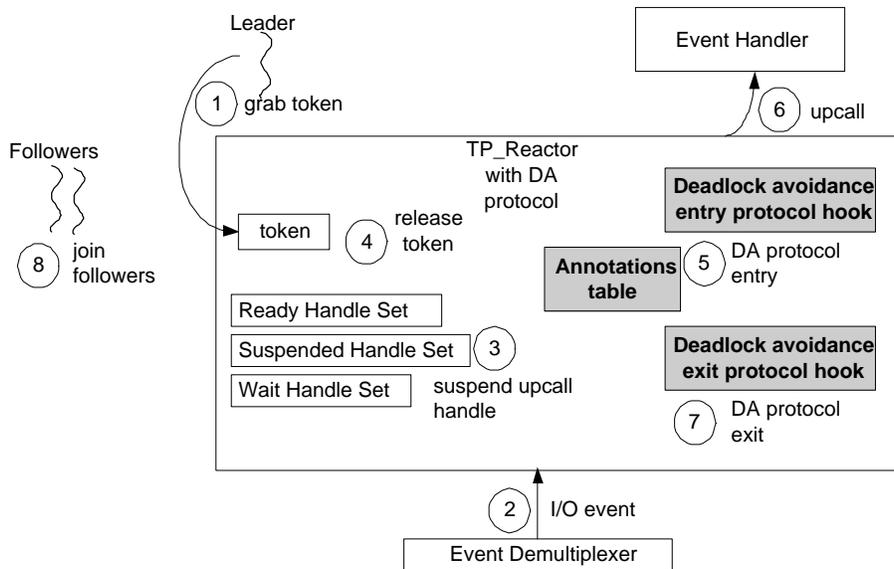
Figure 6. Thread Pool Reactor with Deadlock Avoidance

1. A shared token is used to control access to the reactor. One of the threads from the threadpool acquires the token and becomes the leader thread. This thread then waits in the reactor for I/O events. All the other threads are then follower threads waiting for an opportunity to gain access to the reactor.

2. When an I/O event occurs, the leader thread unblocks from waiting on the reactor's event demultiplexer. The leader thread now has the list of I/O channel handles that are ready for dispatching to their associated handlers.

3. The leader thread then iterates through the list of ready handles and selects one I/O handle to dispatch as a method upcall to the event handler associated with that handle (according to the associations stored in the reactor's handler repository at that time). Before dispatching the upcall, the leader thread suspends the I/O handle associated with the upcall. This is done so that the event handler is not called again in the context of another thread in case the handle becomes ready again while the upcall is already in progress.

4. The leader thread releases the token it has been holding. Consequently, one of the waiting follower threads acquires this token and becomes the leader, hence gaining access to the reactor.

5. The former leader thread now executes the deadlock avoidance protocol. In order to impose as few changes to the existing ACE reactor framework as possible, we used the template method design pattern [22] to introduce hook methods before and after the upcall is made. We then added a new class to ACE called Deadlock_Free_TP_Reactor that overrides those hook methods according to the DA protocol. The call graph annotations for the DA protocol are stored within the Deadlock_Free_TP_Reactor, as a table with an annotation for each of the handlers registered with it. The number of available threads in the thread pool is also stored as a state variable in the Deadlock_Free_TP_Reactor. Based on the specific DA protocol, this state variable is incremented and decremented in the protocol's post-upcall and pre-upcall hook methods respectively, and certain I/O handles other than the upcall handle may be suspended. For example in the BASIC-P protocol [8], all handles whose annotations are less than the number of currently available threads in the thread pool are suspended. By default, these hook method implementations

are empty methods that can be inlined out by an optimizing compiler and hence incur little or no overhead as is quantitatively demonstrated in Section 7.

6. The upcall is made to the event handler.

7. The post-upcall hook method is called, in which the handles that were suspended in the pre-upcall hook method are resumed so that the reactor can demultiplex events for these handles (including the handle associated with the upcall that was just completed).

8. The former leader thread then joins the group of follower threads waiting to acquire the token for access to the reactor.

```
state dispatch_event_handlers;                          state wait_for_handle_input_return;
    provided size(hot_saps_read_set_) > 0;              input handle_input_return(par_context, rc);
(A) next_non_suspended_hot_sap :=                     (F)    call ISS_resume_sap(({Reactor}reactor_).sap_read_set_,
        call ISS_pop_first_non_suspended_sap(hot_saps_read_set_);                 suspended_sap_);
    event_handler :=                                        endif
(B)     call HR_get_handler(({Reactor}reactor_).handler_rep_,  task ({Reactor}reactor_).avail_threads_ :=
                    next_non_suspended_hot_sap);    (G)           ({Reactor}reactor_).avail_threads_ + 1;
(C) call ISS_suspend_sap(({Reactor}reactor_).sap_read_set_,     call ISS_mark_deny_set(({Reactor}reactor_).sap_read_set_,
                    next_non_suspended_hot_sap);                      ({Reactor}reactor_).avail_threads_);
(D) output handle_input(context,                        task ({Reactor}reactor_).state_changed_flag_ := 1;
                    next_non_suspended_hot_sap) to event_handler; (H)  nextstate done;
    task ({Reactor}reactor_).handle_events_in_progress_ :=  endstate;
        ({Reactor}reactor_).handle_events_in_progress_ - 1;
    task suspended_sap_ := next_non_suspended_hot_sap;
    task ({Reactor}reactor_).avail_threads_ :=
(E)     ({Reactor}reactor_).avail_threads_ - 1;
    call ISS_mark_deny_set(({Reactor}reactor_).sap_read_set_,
                    ({Reactor}reactor_).avail_threads_);
    nextstate wait_for_handle_input_return;
```

**Figure 7. Extracts from the IF Model for TP Reactor with Deadlock Avoidance**

Figure 7 shows parts of the IF model for deadlock avoidance protocol support in a Thread Pool (TP) reactor. It should be noted that there is a distinct difference between a single-threaded reactor model and the thread pool reactor model shown in Figure 7. In the former, a single thread uses the reactor to wait on multiple I/O channels. Once a set of SAPs becomes ready, this thread iterates through the set of ready SAPs and dispatches upcalls to each of the corresponding event handlers sequentially. Only after *all* upcalls have been dispatched, does the thread return to the reactor to watch for I/O events again. In contrast, in the TP reactor model the leader thread chooses a non-suspended SAP from among the ready SAPs. The IF procedure ISS_pop_first_non_suspended_sap (A) is used to extract this information from the set of ready SAPs. The leader thread then obtains (B) the corresponding event handler using the HR_get_handler IF procedure. It suspends (C) this SAP using the IF procedure ISS_suspend_sap before making the upcall (D). All these IF procedures are realized using C++ data structures. Note that these procedures take the reactor's SAP set as an inout parameter and hence all modifications made through the C++ data structure are reflected in the SAP set owned by the reactor.

In the TP reactor implementation, a token is maintained to control access to the reactor, as was discussed before. In the model, we use a state variable to control access to the reactor: the handle_events_in_progress_ state variable (E). Each thread checks this variable to make sure that there are no other threads already in the leader role. All the follower threads block on the condition that this variable becomes 0. In the case where multiple threads become eligible for leadership, one thread is chosen non-deterministically.

Figure 7 also illustrates modeling of a specific DA protocol in the context of the TP reactor. (E) and (G) show the entry and exit protocols respectively. The model shown here implements the BASIC-P protocol [8]. The entry protocol decrements the number of available threads by 1. It then uses the IF procedure ISS_mark_deny_set procedure to suspend all the SAPs whose call graph annotations are less than the number of available threads. After the upcall to the appropriate event handler, control returns back to the reactor. Based on the return value, the reactor may deregister the handler, resulting in removal of the handler from the handler
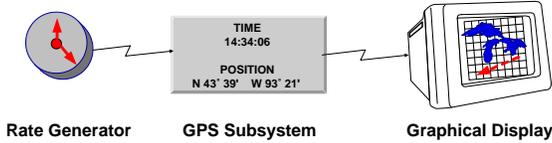
Figure 8. Example DRE Avionics System
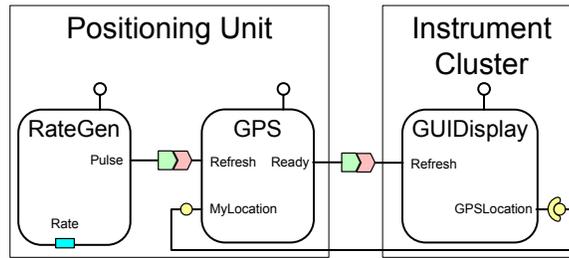
Figure 9. Avionics Example Component Model

repository, or it may resume the SAP that was suspended before the upcall. The exit protocol is then executed. The number of available threads is incremented by 1 and ISS_mark_deny_set is called again to go through the reactor SAP wait set and suspend/resume SAPs based on the new state.

6 Case Study: Distributed Concurrency Scenarios

Section 5 presented a case study in which we showed the ability of our approach to capture nuances of a single sophisticated concurrency control mechanism. In this section, we illustrate how timed automata models can be used to analyze timing and liveness properties in a broader set of scenarios. We first describe a motivating example [23] - a simple DRE system from the domain of avionics mission computing [24] - and show how a high level model of this system can be transformed using our system model described in Section 2 and our modeling and model checking approaches described in Sections 3 and 4 to analyze the timing and liveness properties of the system *taking into account the semantics of alternative middleware configurations with which this system can be implemented.*

Figure 8 shows the elements of our example DRE avionics system: (1) a *Rate Generator*, which wraps a hardware timer and sends periodic events to event consumers that register for those events; (2) a *GPS Subsystem*, which wraps one or more hardware devices for navigation and caches a periodically refreshed location value to provide low-latency response; (3) a *Graphical Display*, which wraps the hardware for a heads-up display device in the cockpit to provide visual information to the pilot and a location value that is updated by querying an interface on the GPS component when the controlling software receives a triggering event.

This example is representative of a class of DRE systems where clusters of closely-interacting components are connected via specialized networking devices, such as VME-bus backplanes. Although the functional characteristics of these systems may differ, they often share the rate-activated computation and display/output timing constraints illustrated here. Figure 9 shows the high level model of the primary software components illustrating both data flow (RateGen to GPS and GPS to Display) and control flow (Display to GPS). Note that there are two models of communication in these systems: (1) a push-model used by the rate generator component to send a triggering event to the GPS subsystem, and by the GPS subsystem to communicate the availability of data, and (2) a pull-model used by the display subsystem to query the location data from the GPS subsystem. The push-model is typically implemented using a publish-subscribe event channel, and the pull-model using a remote function call.

Figure 10 illustrates a high-level model that is used to analyze properties of the system assembly - *e.g.*, compatibility of components, data-flow analysis, and control-flow analysis. This high level model does not include details about the middleware platform on which the system is deployed. Figure 11 illustrates a middleware-level model of the system showing details about behaviorally rich building blocks like reactors, event handlers, and thread pools. Corresponding to each communication channel, there is an event handler. For
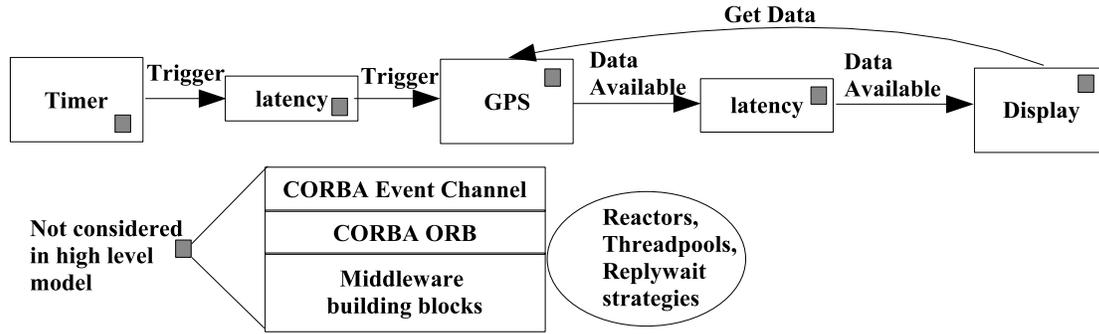
Figure 10. High Level Analysis Model

example, the Timer_EC_EH event handler handles remote requests sent from the RateGen component to the Event Channel(EC), the GPS_EC_EH event handler handles remote requests sent from the GPS component to the EC and so on.
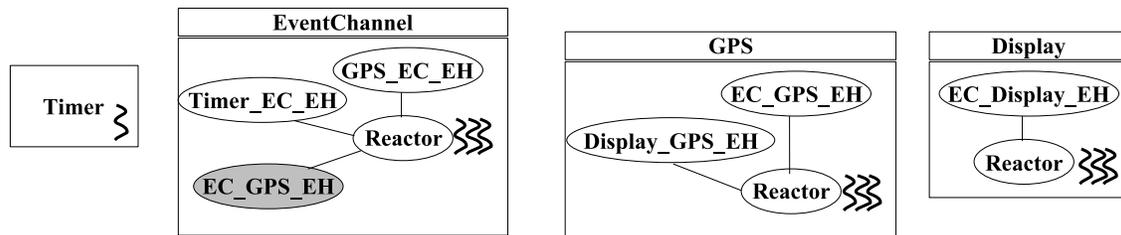
Figure 11. Middleware Level Analysis Model

To illustrate how timed automata models of middleware can be used to analyze timing and liveness properties in practice, we have developed models of four simple but representative example scenarios using the low-level models described in Sections 3 through 5 to capture different segments of the middleware model shown in Figure 11. In each of these scenarios, we vary the semantics of the reactor and event handler models to illustrate how interference can arise for different middleware policy and mechanism choices, and to show how in each case the particular form of interference can be analyzed through model checking. Note that systems of the kinds shown in Figure 8 can be deployed on a variety of implementation platforms, and the lower level models may vary with those choices, *e.g.*, two-way calls, one-way calls with different messaging options [12] (SYNC_WITH_TARGET, SYNC_WITH_SERVER, *etc.*), Asynchronous Messaging Interface [25], Asynchronous Message Handling [26]. Our models capture the semantics of common middleware building blocks with which these different design solutions can be realized, thus giving our approach broad applicability.

Scenario 1 – blocking in a single reactor.   In distributed real-time embedded systems, correct operation can depend on satisfying relatively simple timing constraints such as receiving the result from a method invocation before a relative deadline. In this example, we consider a case where system timing is affected by interference between nominally independent call sequences, when they must contend for shared resources such as the CPU.   We first consider a scenario where a single thread is used by a reactor to demultiplex events to its registered event handlers. The extent to which the event handlers contend for shared resources impacts whether or not a deadline miss can occur. Using our models we can determine (1) whether any deadline misses can occur due to interference between call sequences, and (2) if a deadline miss is possible, sequences of actions that can cause it to occur. For example if the RateGen and

16

GPS components push events at roughly the same time, then whichever event handler (Timer_EC_EH or GPS_EC_EH) is dispatched first will delay the other event handler, potentially resulting in a missed deadline.

Scenario 2 – multiple reactors, *WaitOnConnection* strategy.   In addition to analyzing interference arising from direct contention between handlers for a single resource, it is important to evaluate more complex interference scenarios involving sequences of inter-dependent actions. In this example, we show how timing properties of the system are affected not only by interfering call sequences, but also by the strategy used to wait for replies from remote function calls. For example, consider a situation in which the RateGen component pushes a trigger event to the EC, which in turn pushes that event to a GPS component, which then pushes a dataAvailable event to the EC. After sending the remote function invocation request to the GPS component, the Timer_EC_EH handler waits on that same channel for the reply since the push method is a two-way call in standard CORBA COS Event Service. Because of the interference between the WaitOnConnection reply wait strategy, the topology of the event handler call graph and the use of a single thread in the reactor, deadlock can occur if the single thread in the EC reactor is already in an upcall (to Timer_EC_EH) when there is an incoming remote function call invocation request (for the dataAvailable event) from the GPS component to the EC. We can use IF observers to detect such a deadlock situation by giving an unusually high value (sometimes called a "watchdog timer") for the event handler's deadline and watching for that deadline to be missed, which would indicate that there are no enabled transitions and that time has progressed in effect to infinity (the unusually high value) within the model checker.

Scenario 3 – multiple reactors, *WaitOnReactor* strategy.   The problem raised in Scenario 2 by the WaitOnConnection strategy, in which nesting of calls by the reactor leads to a deadlock preempted call chain, can be alleviated in part through use of an alternative strategy for waiting for the reply from the remote event handler, called *WaitOnReactor*. Using this strategy in Scenario 2, the reactor waits for either the reply (corresponding to the trigger event push) to come back from the GPS, or for another I/O event for Timer_EC_EH or GPS_EC_EH to arrive, and dispatches whichever I/O event arrives first. Waiting for both the pending reply and new request events is done by creating a reply event handler EC_GPS_EH (shown shaded in Figure 11) that is registered with the reactor until the pending reply comes back from GPS and is delivered to EC_GPS_EH. Hence the deadlock in the previous example is avoided, since the single thread in the EC is not only waiting on I/O events on a particular interaction channel (EC-GPS), but rather waits for I/O events on all registered interaction channels (GPS-EC, EC-GPS, Timer-EC). However, this approach in turn introduces further concurrency issues that must be evaluated, which our approach does through model checking.   Consider for example, that when the EC makes a remote call to the GPS component, the EC thread waits on its reactor for the reply (EC_GPS_EH). While waiting for the reply, intervening requests from other suppliers – *e.g.*, the RateGen component – could be processed by the EC's reactor, and this could cause blocking delays in the processing of the reply on which EC_GPS_EH is waiting, possibly leading to a missed deadline.

Scenario 4 – multiple reactors, multiple threads.   The deadlock scenario in Scenario 2 can also be resolved by adding additional reactor threads to the EC. However, adding more threads does not guarantee deadlock freedom in general, since more than one supplier might call the EC concurrently, again leading to deadlock.   Any k threads in the EC reactor can be obtained by k distinct concurrent calls to the EC, leaving no threads to handle the call to GPS_EC_EH and deadlocking each call chain. We provide a more detailed analysis of this particular problem, and of alternative protocols to avoid it, in [8].

To evaluate this scenario, we again used the thread pool reactor model with and without the BASIC-P deadlock avoidance protocol, as in the case study presented in Section 5. Recall that in the BASIC-P protocol, as part of the entry hook the number of available threads in the reactor is decremented by 1, and the number of available threads in the reactor is incremented by 1 as part of the exit hook. Using the complete model built using the multi-threaded reactor model shown in Figure 6 in Section 5.1, we verified three things: (1) there is no deadlock with a single component and ≥ 2 threads in the EC reactor; (2) without a deadlock avoidance protocol there is a deadlock when 2 components try to invoke the same call sequence even if there are 2 threads in the EC reactor; (3) when we then introduce the BASIC-P protocol, we prevent deadlock but introduce blocking delays which in turn need to be modeled to check for deadline misses. For example, with the BASIC-P protocol, one component could be blocked because the EC reactor reserves 2 threads for the call sequence initiated by another component to be able to complete.

7   Analytical and Empirical Evaluations

Table 1. Model Checking Statistics without/with State Space Reduction (SSR), Live Variable Analysis (LVA)

	No SSR						With SSR					
	No LVA			With LVA			No LVA			With LVA		
	#states	#trans	time	#states	#trans	time	#states	#trans	time	#states	#trans	time
Scenario1	531	540	1s	493	503	1s	76	75	1s	76	75	1s
Scenario2 1 flow	226	225	1s	226	225	1s	868	930	1s	868	930	1s
Scenario2 2 flows	229,260	229,284	699s	275,755	275,779	917s	9607	9606	31s	9607	9606	29s
Scenario3	85	84	1s	85	84	1s	19	19	1s	19	18	1s
Scenario4 3 flows, No DA	*	*	*	*	*	*	1460	1501	7s	1447	1488	7s
Scenario4 3 flows, BASIC-P	*	*	*	*	*	*	74,453	74,452	453s	74,323	74,323	412s

The results of running exhaustive simulations in IF for each of the four scenarios described in Section 6 are shown in Table 1. These exhaustive simulations were run on a Pentium 3 933Mhz processor with 512MB RAM. For all the runs, we used the partial order reduction and depth-first-search options in the IF exhaustive simulator, as was suggested by the IF tool developers. For each scenario, we ran the exhaustive simulation with and without the state space reduction (SSR) strategies described in Section 4. We also compared the results with and without live-variable analysis (LVA), which can be used to eliminate dead variables.

For Scenarios 1 and 2, the state space reduction strategies provide noticeable benefits. For Scenario 3, the results show that although the state space reduction strategies suffer some overhead when the number of clients is small, they are beneficial when the number of clients (flows) increases. For scenario 4, there was a state space explosion without SSR (indicated by asterisks in Table 1), whereas the model exploration became quite tractable with SSR, although the state space does increase when the deadlock avoidance protocol is present, showing that there are potential trade-offs in the checking required for one property (blocking factors) when checking for another property (deadlock) is avoided through use of a proved protocol [8].

7.1   Deadlock Avoidance Overhead

We conducted empirical evaluations of the overhead of the deadlock avoidance protocol whose implementation was discussed in detail in Section 5. Our experimental setup is illustrated in Figure 12. We used an ACE thread pool reactor watching a set of

ACE_Pipes with only one thread in the threadpool. We used a unique event handler corresponding to each of the pipes. The event handlers do not make any remote function calls, hence the height annotation for each of the event handlers according to the protocol is 1. This is sufficient for measuring the overhead of the protocol implementation within the threadpool reactor because even if the height annotations are different, the mechanisms (the hook functions discussed in Section 5) for protocol execution are still the same.
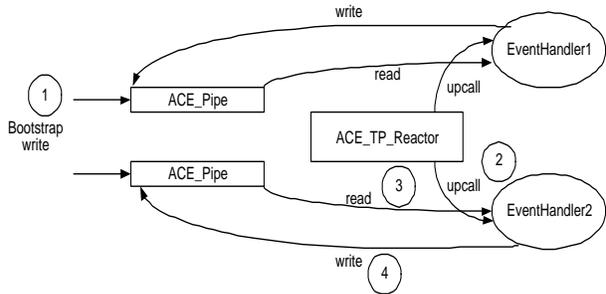

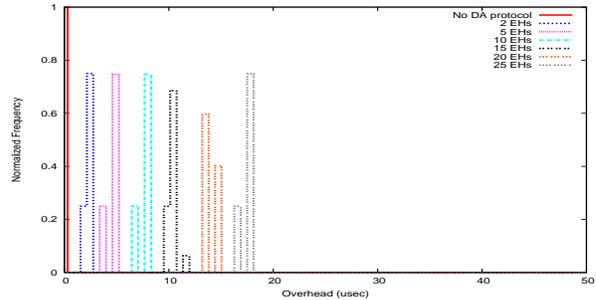
Figure 12. DA Protocol Experiment Setup

Figure 13. DA Protocol Overhead

To bootstrap the experiment, we wrote a constant-sized buffer of bytes to each of the pipes (1). The reactor demultiplexed the events to the appropriate event handlers. On an upcall (2) from the reactor, each event handler read (3) from one end of the pipe and wrote (4) to the other end of the same pipe. In other words, each event handler "fed" itself data. This setup makes it easy to increase the number of event handlers and see the direct effect of that increase on the protocol execution time, since the number of event handlers alone determines the time taken by the reactor to suspend a set of event handlers before making an upcall. We ran these experiments, and the experiments described in Section 7.2, on a Pentium 3 1.4Ghz machine with 1GB RAM. For all of the experiments, we used ACE version 5.4.7, the KUSP Libertos [27] Linux 2.6.12 based kernel, and the Data Streams Kernel Interface (DSKI) [28] and Data Streams User Interface (DSUI) frameworks for instrumentation and processing of collected data.

Figure 13 shows the overhead of protocol execution for 2, 5, 15, 20 and 25 event handlers, each with an annotation of 1 and a single thread in the threadpool. As we expected, the time taken was shown to be dependent on the number of event handlers, since the protocol implementation suspends all event handlers except one before making an upcall to that event handler. It is significant that without a deadlock avoidance protocol there was no measured overhead, which was our original goal in making the default protocol hook functions empty methods which can be inlined away resulting in little or even no overhead for use cases that don't use a deadlock avoidance protocol.

7.2 Blocking Behavior

In this section, we present an empirical evaluation of the blocking behaviors in scenarios 1, 3, and 4 discussed in Section 6. Blocking delays affect the timeliness of the system, and account for deadline misses which were found by exhaustive simulation of the IF-models. For each experiment, we identified a set of operating system handles associated with event handlers, and for each of these handles we measured the delay between the time when a message was ready to be read on that handle and the time when an event handler actually started processing that message (handle_input).

A socket layer DSKI instrumentation point (EVENT_SOCK_DEF_READABLE DSKI event) was used to log an event whenever a buffer of bytes was enqueued into a socket queue (this instrumentation point is in the *kernel/sock.c: sock_def_readable* function of the Linux 2.6.12 kernel). This function is called by the network-protocol-specific (*e.g.*, TCP/UDP/Unix-sockets) code after queuing bytes

into a socket queue. This measure increases the accuracy of our measurements when compared to the alternative approach where we might measure the interval between the time when the message was sent by a client and the same message read by an event handler. The problem with the latter approach is that the measured delay may also include the propagation delay of the message, which might skew estimation of the actual blocking delay that is caused by interleaving calls to the same reactor. To enable correlation between the EVENT_SOCK_DEF_READABLE DSKI event and user space DSUI events, we recorded the socket handle identifier along with the appropriate DSUI events (*e.g.*, HANDLE_INPUT in an event handler). We also modified the kernel for these experiments to include the socket handle as part of the socket data structure in the kernel so that this information was available during logging of the DSKI event and could be used during post processing to correlate kernel and user events.
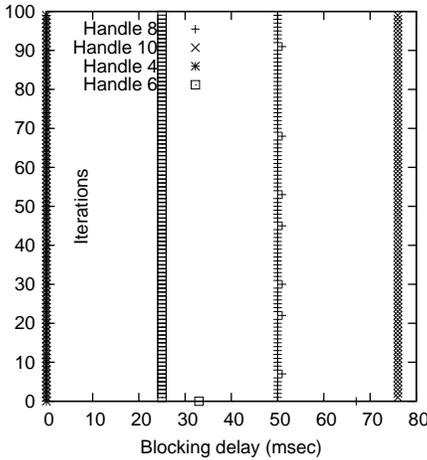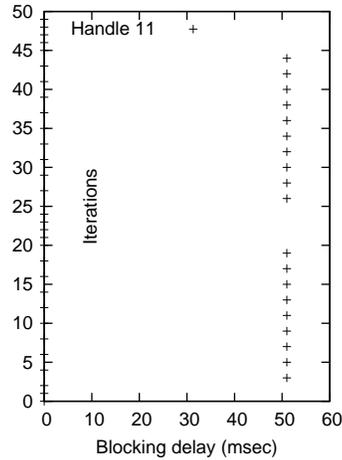

Figure 14. Scenario 1
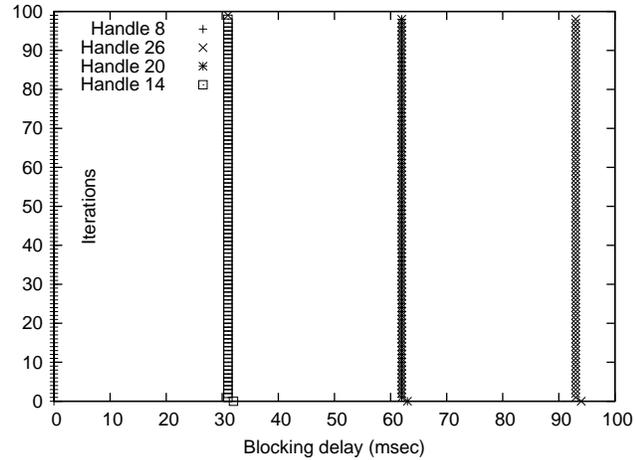

Figure 15. Scenario 3

Figure 16. Scenario 4

Figure 14 shows the blocking factors in Scenario 1 with 4 suppliers and 4 corresponding event handlers all using the same reactor. The execution time for each event handler is 25ms. We used an artificial CPU-bound load to simulate application specific computation done by the event handlers. The results show that with a single thread, some of the clients suffered blocking delays which could result in missed deadlines. The blocking delay at a particular socket handle depends on the number of interleaving calls concurrently passing through the reactor. In this scenario, clients could suffer blocking delays of 25ms, 50ms and 75ms, corresponding to the execution of 1, 2, or 3 event handlers respectively, before the message at that socket is picked up by the corresponding event handler.

Figure 15 shows the blocking factors for Scenario 3 with 3 event handlers. The Timer_GPS_EH event handler registers a separate reply handler and then the thread waits in the reactor until it gets a reply from the remote event handler. Figure 15 shows that the processing the reply *may* be delayed by other event handlers processing at that reactor. In this scenario, the reply processing was delayed at most by the execution time of one other handler, 50ms.

Figure 16 shows the blocking behavior introduced by the BASIC-P deadlock avoidance protocol. We ran the experiment with 3 threads in the first reactor and 2 threads in the second reactor, with both reactors implementing the BASIC-P deadlock avoidance protocol. The execution times of all the event handlers was fixed at 10ms. From the results, we can see that there is no deadlock, but there are blocking delays. We measured the blocking delays at the socket handle corresponding to Timer_EC_EH in the first reactor. One of the suppliers (timer) was delayed for about 30ms, which is the execution time for one nested call traversing through Timer_EC_EH, EC_GPS_EH and GPS_EC_EH. Another supplier was delayed for about 60ms (execution time for completion of 2 nested calls) and a third one was delayed for about 90ms (execution time for completion of 3 nested calls).

20

8 Related Work

Model-integrated computing. Integration of DRE systems using different components demands a great deal of *a priori* modeling and analysis. The key theme of Model Integrated Computing (MIC) [29] is that it extends the scope and usage of models so that they form the "backbone" of a model-integrated system development process. The Generic Modeling Environment (GME) [30] is a configurable toolkit for creating domain-specific modeling and software synthesis environments. Ptolemy II [31] is another modeling environment for embedded systems that provides a rich set of computation models including the Giotto model [32] that provides an abstract infrastructure model for the implementation of embedded control systems with hard real-time constraints.

Our research fits within the Model-driven Middleware (MDM) [33] paradigm which applies model-based techniques such as MIC to the domain of middleware. Moreover, our approach provides a more rigorous basis for middleware verification than current model-based middleware configuration techniques. We also plan to investigate the suitability of integrating our formal models within the GME and Ptolemy environments. Unlike the Giotto approach which creates a specialized concurrency environment for enforcement of timing properties, our approach is to model canonical *existing* fine-grain middleware abstractions found in common use, as a basis for evaluation and composition of those elements.

Model-driven middleware. CADENA [34] is an integrated environment for building and modeling CORBA Component Model (CCM) [23] systems. The CoSMIC [33] tool set supports integrated model-driven component assembly, deployment and configuration. [35] shows how model checking using the extensible Bogor [14] model checker has been applied to verifying event-driven systems using an event channel. The low-level formal models we are developing, combined with the middleware mechanisms our models represent potentially could be integrated with these tool sets to provide fine-grained model checking and software synthesis capabilities over a common and reusable software base.

Formal techniques in CORBA. [36] introduces new stereotypes in UML and describes ways to map these stereotypes to a process algebra (FSP) and then use model checking to detect deadlocks. In addition to the single threaded synchronous request scenario discussed in that paper, which results in deadlock, in this paper we consider similar scenarios which may or may not result in deadlock depending on the strategies used to wait for a reply in the underlying middleware infrastructure. Moreover, we also check the timing properties of the system. Kamel's work [37] uses model checking to verify the GIOP protocol used in CORBA based systems. Duval's work [38] also uses a model checking approach to verifying CORBA based systems. [39] uses a formal language named TRIO to specify CORBA-based distributed applications and uses a proof-based approach for reasoning about systems and verifying their correctness. Our work differs from those approaches in that it provides executable formal models for *fundamental middleware building blocks* that can be used to verify a variety of middleware services including, but not limited to, CORBA implementations.

Fine-grain middleware building blocks. Our work so far has focused on mechanisms in ACE. Our techniques are applicable to other environments where abstractions like the Selectors in Java NIO [40] are similar to the reactor and event handler models we have already developed. Moreover, our modeling approach has potential application to other less similar environments, *e.g.*, to model and analyze the fundamental building blocks provided by platforms such as TinyOS [41].

9 Conclusions and Future Work

Our middleware modeling approach presented in this paper is designed to address the need for a more formal basis for verification of correct middleware construction and configuration in the context of individual applications. The examples presented in Sections 5 and 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; (3) alternative strategies for handlers relinquishing control, such as WaitOnConnection and WaitOnReactor; and (4) multiple additional on-line protocols, *e.g.*, for deadlock avoidance(DA), real-time admission control, or security authorization. Furthermore, the constraints each application places on timing and other properties will alter the criteria by which system timeliness and liveness are evaluated.

The results of our simulations and experiments presented in Section 7 motivate the need for detailed modeling of low-level middleware mechanisms, and evaluation of those models through model checking tools. Our model checking results show that with a deadlock avoidance protocol the size of the state space increases. However, the DA protocols guarantee deadlock freedom under certain conditions, and in some cases analysis can be used to determine a number of threads in each reactor that would avoid deadlock without use of a run-time DA protocol. Model checking can be used to verify whether there are any deadline misses in the system resulting from a variety of blocking factors. With or without DA protocols, our models can be used for model checking behavior of systems built using the middleware primitives we have modeled. Therefore, the results of our evaluations support our contention made in Section 3 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 work presented in this paper motivates the suitability and viability of that approach.

The goal of our research is to address the problem of evaluating complex middleware environments, while preserving both rigor in analysis and tractability in applying our approach to real world systems. To meet that goal our future work focuses on developing an ever-expanding set of robust, modular, and composable models of middleware building blocks, and integrating those models within model-integrated computing tool sets such as those described in Section 8. We will also continue our work on formally verified efficient protocols [8], along with the other optimizations described in Section 4 to both expand the expressive power and reduce the burden of model checking. Since our models are executable models, they can also be used to run guided simulations to verify specific scenarios in cases where exhaustive state space exploration of all possible scenarios is intractable.

References

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

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

[3] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems," in *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, (San Francisco, CA), IEEE, Mar. 2005.

[4] D. I. E. Office, "Model-Based Integration of Embedded Software (MoBIES)." www.darpa.mil/ixo/mobies.asp.

[5] E. Turkay, A. Gokhale, and B. Natarajan, "Addressing the Middleware Configuration Challenges using Model-based Techniques," in *Proceedings of the 42nd Annual Southeast Conference*, (Huntsville, AL), ACM, Apr. 2004.

[6] 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.

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

[8] 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.

[9] C. Sanchez, H. B. Sipma, V. Subramonian, C. Gill, and Z. Manna, "On Efficient Distributed Deadlock Avoidance for Real-Time and Embedded Systems," in *Submitted to IPDPS 2006*, 2006.

[10] 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.

[11] 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*, vol. 91, July 2003.

[12] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 3.0.2 ed., Dec. 2002.

[13] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on uppaal," in *SFM*, pp. 200–236, 2004.

[14] 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), ACM, Sept. 2003.

[15] G. J. Holtzman, "The Model Checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, pp. 279–295, May 1997.

[16] M. Bozga, S. Graf, I. Ober, and L. Mounier, "IF-2.0: A validation environment for Component-Based Real-Time Systems," in *Proceedings of CAV'02*, Springer-Verlag LNCS 2404, 2002.

[17] M. Bozga, J. Fernandez, L. Ghirvu, S. Graf, J. Krimm, and L. Mounier, "IF: A Validation Environment for Timed Asynchronous Systems," in *Proceedings of CAV'00*, 2000.

[18] S. Bornot, J. Sifakis, and S. Tripakis, "Modeling urgency in timed systems," in *COMPOS*, pp. 103–129, Springer-Verlag LNCS 1536, 1997.

[19] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Boston: Addison-Wesley, 2002.

[20] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Reading, Massachusetts: Addison-Wesley, 2002.

[21] 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.

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

[23] N. Wang, C. Gill, D. C. Schmidt, and V. Subramonian, "Configuring Real-time Aspects in Component Middleware," in *Proc. of the International Symposium on Distributed Objects and Applications (DOA'04)*, (Agia Napa, Cyprus), Oct. 2004.

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

[25] D. C. Schmidt and S. Vinoski, "Introduction to CORBA Messaging," *C++ Report*, vol. 10, November/December 1998.

[26] M. Deshpande, D. C. Schmidt, C. O'Ryan, and D. Brunsch, "Design and Performance of Asynchronous Method Handling for CORBA," in *Proceedings of the 4th International Symposium on Distributed Objects and Applications*, (Irvine, CA), OMG, October/November 2002.

[27] Douglas Niehaus, *et al.*, "Kansas University Real-Time (KURT) Linux." `www.ittc.ukans.edu/kurt/`, 2004.

[28] D. Niehaus, "Improving support for multimedia system experimentation and deployment," in *Workshop on Parallel and Distributed Real-Time Systems*, (San Juan, Puerto Rico), Apr. 1999. Also appears in Springer Lecture Notes in Computer Science 1586, Parallel and Distributed Processing, ISBN 3–540–65831–9, pp 454–465.

[29] J. Sztipanovits and G. Karsai, "Model-Integrated Computing," *IEEE Computer*, vol. 30, pp. 110–112, Apr. 1997.

[30] 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.

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

[32] T. A. Henzinger and C. M. Kirsch, "The embedded machine: predictable, portable real-time code," *SIGPLAN Not.*, vol. 37, no. 5, pp. 315–326, 2002.

[33] 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), ACM, Nov. 2002.

[34] 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.

[35] 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.

[36] N. Kaveh and W. Emmerich, "Validating distributed object and component designs," in *SFM*, pp. 63–91, 2003.

[37] M. K. A1 and S. Leue, "Formalization and validation of the general inter-orb protocol (giop) using promela and spin," in *Int. Journal on Software Tools for Technology Transfer*, Springer–Verlag, 2000.

[38] G. Duval, "Specification and verification of an object request broker," in *ICSE '98: Proceedings of the 20th international conference on Software engineering*, (Washington, DC, USA), pp. 43–52, IEEE Computer Society, 1998.

[39] A. Coen-Porisini, M. Pradella, M. Rossi, and D. Mandrioli, "A formal approach for designing corba-based applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, no. 2, pp. 107–151, 2003.

[40] R. Hitchens, *Java NIO*. O'Reilly, 2002.

[41] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pp. 93–104, ACM Press, 2000.