Report Number: WUCSE-2005-29

2005-04-20

Composable Timed Automata Models for Real-Time Embedded Systems Middleware

Venkita Subramonian, Christopher Gill, Cesar Sanchez, and Henny Sipma

Middleware for distributed real-time embedded (DRE) systems has grown more and more complex in recent years, to address functional and temporal requirements of complex real-time applications. While current approaches for modeling middleware have eased the task of assembling, deploying and configuring middleware and applications, a more formal, fundamental and lower-level set of models is needed to be able to uncover subtle safety and timing errors introduced by interference between computations, particularly in the face of alternative concurrency strategies in the middleware layer. In this paper, we examine how formal models of lower-level middleware building blocks provide an appropriate level... **Read complete abstract on page 2.**

Composable Timed Automata Models for Real-Time Embedded Systems Middleware

Venkita Subramonian, Christopher Gill, Cesar Sanchez, and Henny Sipma

Complete Abstract:

Middleware for distributed real-time embedded (DRE) systems has grown more and more complex in recent years, to address functional and temporal requirements of complex real-time applications. While current approaches for modeling middleware have eased the task of assembling, deploying and configuring middleware and applications, a more formal, fundamental and lower-level set of models is needed to be able to uncover subtle safety and timing errors introduced by interference between computations, particularly in the face of alternative concurrency strategies in the middleware layer. In this paper, we examine how formal models of lower-level middleware building blocks provide an appropriate level of abstraction for both modeling and synthesis of a variety of kinds of middleware from these building blocks. When combined with model checking techniques, these formal models can help developers in constructing correct combinations and configurations of middleware mechanisms, for each particular application.

2005-29

Composable Timed Automata Models for Real-Time Embedded Systems Middleware

Authors: Subramonian, Venkita; Gill, Christopher; Sanchez, Cesar; Sipma, Henny

April 20, 2005

Abstract: Middleware for distributed real-time embedded (DRE) systems has grown more and more complex in recent years, to address functional and temporal requirements of complex real-time applications. While current approaches for modeling middleware have eased the task of assembling, deploying and configuring middleware and applications, a more formal, fundamental and lower-level set of models is needed to be able to uncover subtle safety and timing errors introduced by interference between computations, particularly in the face of alternative concurrency strategies in the middleware layer.

In this paper, we examine how formal models of lower-level middleware building blocks provide an appropriate level of abstraction for both modeling and synthesis of a variety of kinds of middleware from these building blocks. When combined with model checking techniques, these formal models can help developers in constructing correct combinations and configurations of middleware mechanisms, for each particular application.

Composable Timed Automata Models for Real-Time Embedded Systems Middleware

Venkita Subramonian and Christopher Gill
{venkita,cdgill}@cse.wustl.edu
Department of Computer Science and Engineering
Washington University, St. Louis, MO

César Sánchez and Henny Sipma
{cesar,sipma}@cs.stanford.edu
Computer Science Department
Stanford University, Stanford, CA

Abstract

Middleware for distributed real-time embedded (DRE) systems has grown more and more complex in recent years, to address functional and temporal requirements of complex real-time applications. While current approaches for modeling middleware have eased the task of assembling, deploying and configuring middleware and applications, a more formal, fundamental and lower-level set of models is needed to be able to uncover subtle safety and timing errors introduced by interference between computations, particularly in the face of alternative concurrency strategies in the middleware layer.

In this paper, we examine how formal models of lower-level middleware building blocks provide an appropriate level of abstraction for both modeling and synthesis of a variety of kinds of middleware from these building blocks. When combined with model checking techniques, these formal models can help developers in constructing correct combinations and configurations of middleware mechanisms, for each particular application.

1 Introduction

Significant research has been conducted over the past decade to make middleware more flexible and customizable through the use of pattern-oriented software frameworks [1, 2]. Although this research has increased the flexibility and applicability of middleware to different kinds of applications, managing the choices for customization available to the application developer has become an increasing concern.

To allow middleware to be configured and customized to meet the stringent demands of different distributed real-time embedded (DRE) applications, a body of ongoing research has focused on applying model-driven techniques to developing QoS-enabled middleware. While current approaches for modeling middleware focus on easing the task of assembling, deploying and configuring middleware and middleware-based applications, a more fine-grained formal basis for correct middleware construction and configuration *for each application* is needed.

Formal models have been used to uncover high-level design flaws early in system development. However, these models often are not adequately extended and refined throughout the entire development lifecycle. For example, decisions regarding the deployment of application components or the choice of middleware mechanisms are not reflected in the high-level model. This may result in subtle safety and timing errors being introduced by unexpected *interference* between computations, particularly in the face of alternative middleware concurrency strategies. Therefore, a more foundational set of formal models is needed to be able to uncover such errors.

In this research, we are developing formal models of lower-level middleware building blocks that are fundamental to a variety of kinds of middleware. These lower-level models can then be composed with higher-level formal models to provide a more *complete* model of a system. Our ultimate goal is that these composed models then can be validated for correctness, with high fidelity to the system itself, using model checking techniques. Meeting this goal at a realistic scale will require significant further work, to which the research described in this paper contributes. Specifically, our approach assists in composing correct combinations of middleware mechanisms, and configuring those mechanisms for each specific application. Our approach has the following major impacts: (1) it adds rigor to the model-based approaches to middleware development currently being pursued by the systems research community; (2) it provides high fidelity composable models of foundational middleware building blocks, to the formal modeling community.

Approach: Our approach hinges on the idea that *interference* occurs when the activities of multiple components in a system affect each other in ways that may produce adverse consequences for the system. The notion of *interference* is already well-known in the field of programming languages, where an interference graph over program variables whose live-ranges overlap [3] is used by compilers for register allocation, to minimize the total number of registers needed.

In this research, we extend the notion of *interference* to address concurrency and timing issues crucial to real-time systems. Just as interference graphs are used to analyze interference between *variables* in programming languages, we use formal models to analyze how interference between *computations* in real-time embedded systems impacts the timing behavior of these computations. Specifically, in this research, we examine how different forms of interference produced by different middleware mechanisms can be modeled and analyzed formally.

Formalization: We formalize our definition of interference in real-time embedded systems as follows. Let $C = \{C_1, C_2, ..., C_N\}$ where $N > 0$, denote a set of concurrently running computations in an application. Let R be the set of available resources, a subset of which is used by each of the computations in C. Note that these resources may be acquired and released repeatedly by computations over a period of time. Let T represent a continuous time domain over the set of positive real numbers. Let $U : C \times T \rightarrow 2^R$ be a function that defines the resource usage of a computation over time. Note that $U(C_i, t) \subseteq R$, $(C_i \in C, t \in T)$. Two computations C_i and C_j $(i, j <= N$ and $i \neq j)$ *interfere* with each other if $U(C_i, t) \cap U(C_j, t) \neq \emptyset$. That is, if two computations need the *same* resource at the same time, then there is *interference* between them.

Models: This definition of interference guides our selection of models for analysis of timing and concurrency. In general, we use model checking to ensure soundness. Due to the potential size of the state spaces that need to be checked, we are currently pursuing several optimizations: (1) building highly modular models, by sub-dividing them into fine-grain composable automata; (2) migrating our models from static model checkers to 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 techniques thus reducing the state space that must be explored through model checking.

The rest of this paper is structured as follows. Section 2 presents our solution approach. Section 3 gives examples of how *interference* may be caused by different middleware concurrency and communication strategies, and in turn affect system safety and liveness properties. For each example, we describe how safety and liveness can be analyzed using composable formal models. Section 4 describes related work and Section 5 offers concluding remarks and summarizes future work.

2 Formal Modeling Approach

Our approach addresses the following challenges:

1. Middleware should provide common abstractions that can be re-used across different applications.
2. It should then be possible to make fine-grained modifications or select appropriate configurations to tailor the middleware to the requirements of each specific application.
3. The middleware should allow flexible integration of mechanisms to resolve *interference* between middleware and/or application components.
4. The application developer should be able to validate the correctness of the customized middleware in the context of the application, both formally and rigorously.

These challenges can be resolved by taking a principled and formal model-driven approach to middleware development, using composable models of common middleware building blocks. We contend that the activities of modeling and engineering should be done in an integrated manner rather than as disparate activities. The insights obtained from modeling should be made available and used while making engineering decisions and vice-versa, thus producing the following benefits:

- More complete and detailed models of systems formed by composing application and infrastructure models can make the overall model more faithful to the actual system.
- Safety and liveness properties can be verified automatically.
- Formal models offer more rigorous documentation of middleware engineering expertise currently represented less formally, *e.g.*, as *patterns* [4].
- Formal approaches to correct construction of fine-grain middleware elements enables future work on tools for correct construction of entire systems.

High fidelity modeling of DRE middleware: 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 implementation. For example, distribution middleware such as CORBA [5] object request brokers (ORBs) provides a level of abstraction that is needed for portability and reusability and hence makes an appealing candidate for formal modeling.

Since 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 [6] is a useful and necessary step toward model-driven construction of DRE systems. We contend, however, that to evaluate systemic issues such as safety and liveness, which are crucial to many DRE systems, fine-grained models of lower-level middleware building blocks are needed to analyze crucial details related to concurrency and timing properties.

Results of our previous experience with system software construction indicate the efficacy of such a fine-grained approach. In that work we built a special-purpose ORB called *nORB* [7], with support for real time operation dispatching in the context of memory constrained networked embedded systems. We took a fine-grained bottom-up approach to the development of nORB, starting with lower level elements of the ACE [1] framework: Reactor, Acceptor, Connector, CDR Stream, *etc.* Along with taking a fine-grained approach to building nORB, we used the application itself as a guide for making tradeoffs (*e.g.*, between feature richness and footprint). For example, we restricted the supported sets of data types and message formats to those required by the target application. That work has given us insights into application-driven construction and customization of DRE middleware for this and other domains, allowing us to define composable models with a high degree of fidelity to how DRE middleware is built in practice.

Tools and formalisms for modeling: We use timed automata [8] to model fine-grained DRE middleware mechanisms. We use a model-checking [9] approach to verify safety and liveness properties. To illustrate the kinds of fine-grained models

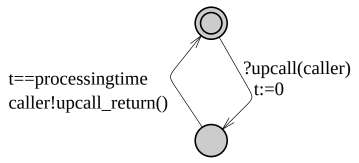

Figure 1. Simplified Event Handler Model

used in our approach, Figures 1 and 2 show simplified formal models developed in the UPPAAL [10] model checker, for two middleware mechanisms: an event handler and a reactor, respectively. An event handler receives a system event through invocation of one of its methods, processes the event, and the method call then returns. A *reactor* is an event dispatching mechanism used in distributed systems to deliver events arriving from multiple sources to event handlers running in one or more threads. For example, reactors can be used in ORBs to demultiplex and dispatch incoming method invocation requests and replies from sockets connected to peer ORBs. Event handlers (*e.g.*, request and reply handlers) are registered with each reactor. The reactor uses a synchronous event demultiplexer (*e.g.*, the UNIX *select* system call), to wait for messages to arrive from one or more ORBs. When a message arrives, the synchronous event demultiplexer notifies the reactor, which then dispatches the appropriate event handler based on the event source.

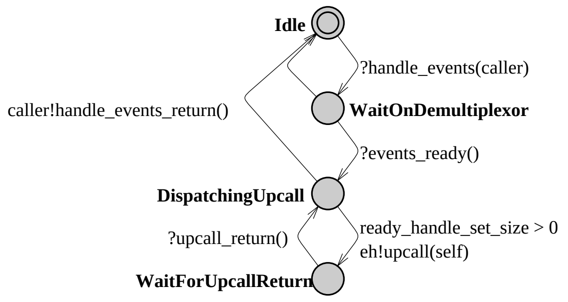

Figure 2. Simplified Reactor Model

3 Illustrative Examples

To illustrate how timed automata models can be used to analyze safety and liveness properties in practice, we now consider four simple but representative example scenarios. In each of these scenarios, we vary the semantics of the reactor and event handler models to illustrate interference 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.

Example 1 – blocking in a single reactor: In real-time embedded systems, safety properties can involve timing constraints such as receiving the result of 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 resources such as CPU cycles. Figure 3 shows two call sequences in which Client1 and Client2 invoke methods and receive replies from event handlers EH1 and EH2 respectively – Client1 and Client2 have relative deadlines at 6 and 8 msec after sending the invocation request message, by which they must receive their respective reply messages.

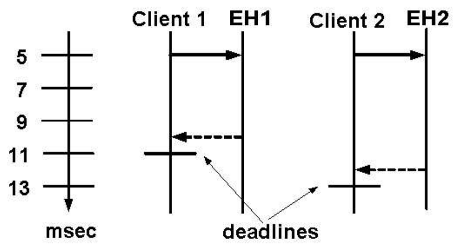

Figure 3. Timeline for Example 1

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 the two call sequences, and (2) if a deadline miss is possible under what conditions it can occur. For example, if both EH1 and EH2 are deployed on the same single-threaded reactor, as shown in Figure 4, then they can only handle events sequentially. If Client1 and Client2 send messages to EH1 and EH2 respectively at roughly the same time, then whichever event handler is dispatched first will delay the other event handler, potentially resulting in a missed deadline.

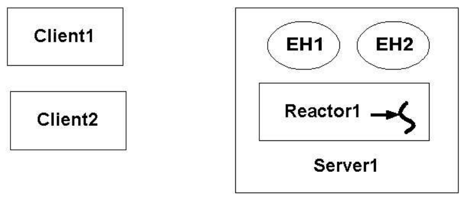

Figure 4. Example 1 Deployment

Figure 5 shows the system model for this example composed in UPPAAL from our fine-grained models. Several features of the model for Reactor1 are worth noting. First, the Reactor1 automaton contains separate transition branches for dispatching to EH1 and EH2. Second, the Reactor1 automaton communicates with the EH1 and EH2 automata via UPPAAL rendezvous annotations, to model calls from the reactor to the event handlers. Third, state variables are used to model a single thread of execution by making the dispatching of event handler calls sequential.

We can then use the composed model to analyze whether there are any deadline misses, by checking temporal logic expressions such as "E◇ Client1.DeadlineMiss or Client2.DeadlineMiss" (whether there is any state in the state space where the Client1 automaton or the Client2 automaton is in its DeadlineMiss state). If this property is true, then the model checker can generate a trace of the call sequence that leads to the property becoming true. For example, Figure 6 depicts a trace generated by UPPAAL that shows a call sequence leading to a deadline miss for Client1. The call sequence shows that the single thread in the reactor first processes the request from Client2 and makes the upcall to EH2. This introduces a blocking delay in the processing of the request from Client1. Using UPPAAL we also verified that there is no missed deadline if only one of Client1 and Client2 is hosted in the reactor. This shows that the combination of the two call sequences in the same thread is what causes a missed deadline.

Example 2 – multiple reactors, *WaitOnReactor* strategy: In addition to analyzing interference arising from direct contention between handlers for a resource, it is important to evaluate more complex interference scenarios involving chains of interdependent handlers. For example, consider a scenario where EH1 and EH2 are deployed on different single-threaded reactors, and a third event handler, EH3, is deployed on the same reactor as EH1 as shown in Figure 7.

In this example, we show how timing properties of the system are affected not only by interfering call sequences, but also the type of strategies chosen to wait for replies from remote calls in a distributed system. For example, if EH1 depends on EH2 for part of its processing, then EH1 would send a message to EH2 on Server 2, and wait for a reply from EH2 before completing its processing and sending a reply back to the client that sent it a message. For purpose of illustration, consider a scenario in which Client 1 invokes EH1 and Client 2 invokes EH3 with relative deadlines of 8 and 7 msec, respectively.

Figure 8 shows the two call sequences for this scenario, one of which goes through a single reactor and the other going through two reactors. Each of these reactors has a single thread. When

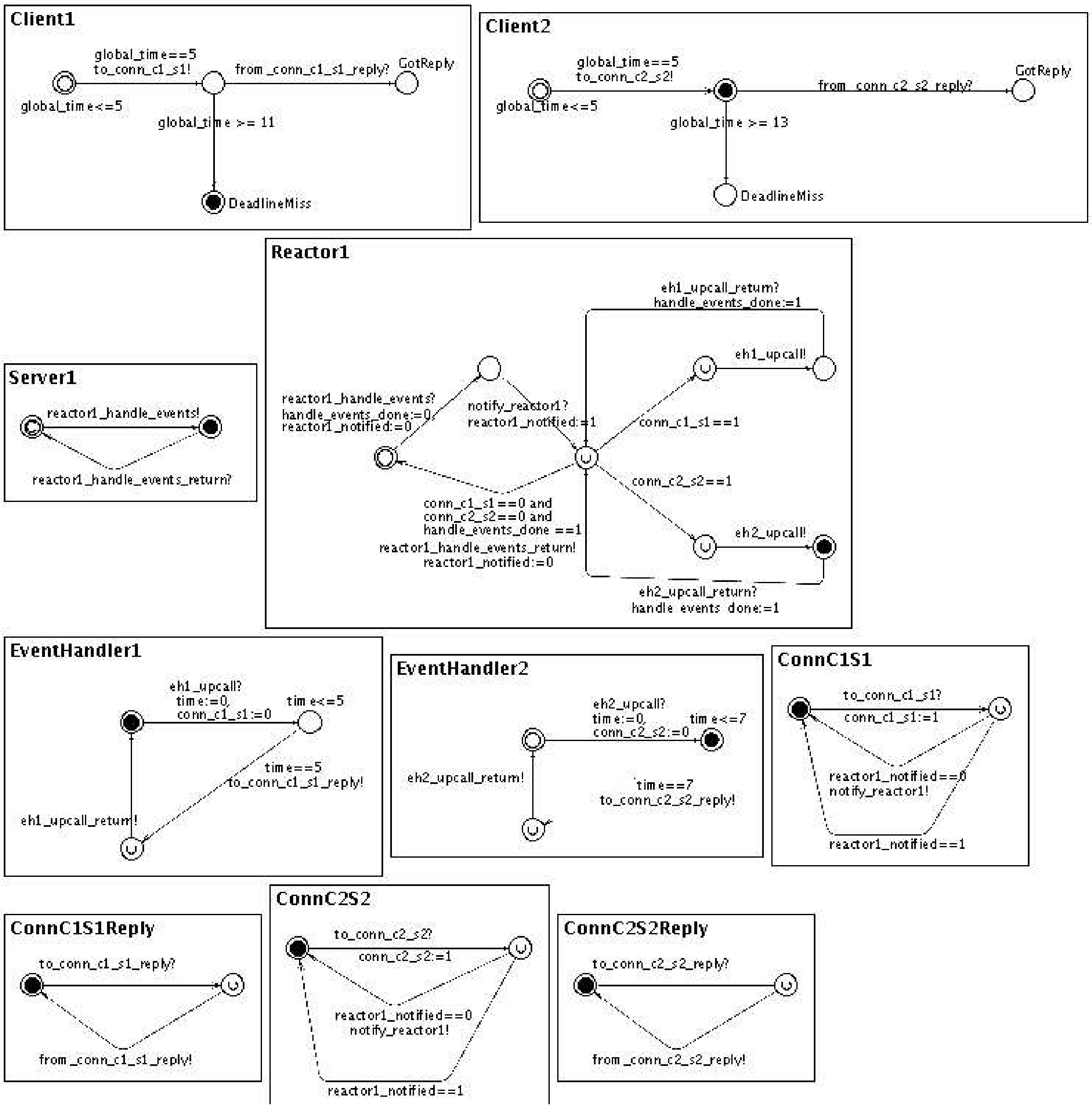

Figure 5. Example 1 - Composed System Model in UPPAAL

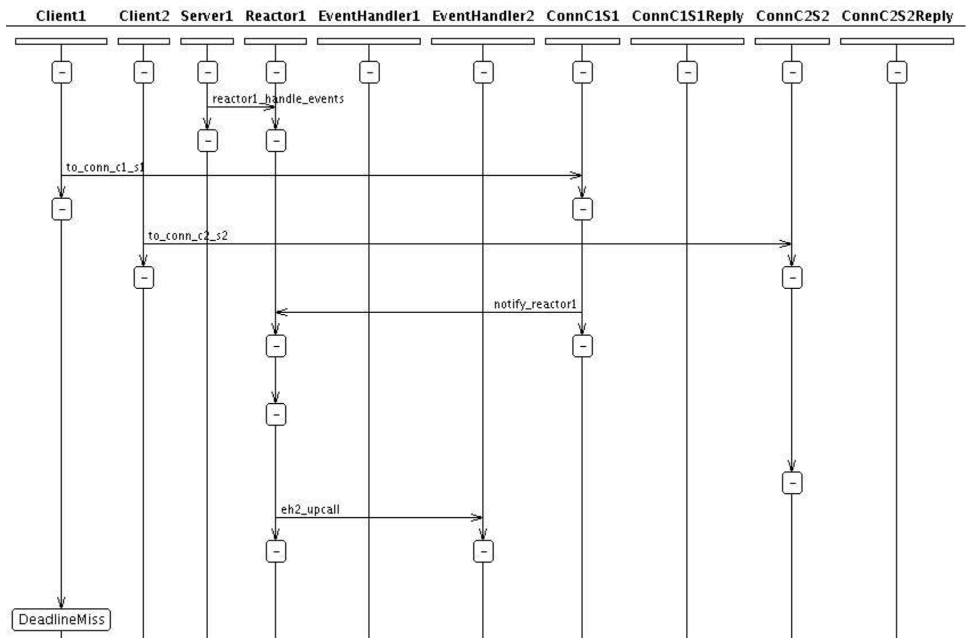

Figure 6. Example 1 Trace - Missed Deadline

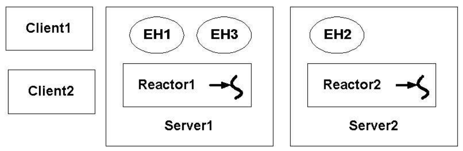

Figure 7. Deployment for Examples 2 and 3

EH1 makes a remote call to EH2, it waits on the reactor (Reactor1) for the reply. While waiting for the reply, there could be interleaving requests that get processed by the reactor. This causes blocking delays in the processing of the reply that EH1 is waiting on. In this example, we can use Client2 to introduce an interleaving call to EH3, while EH1 is waiting for its reply from EH2.

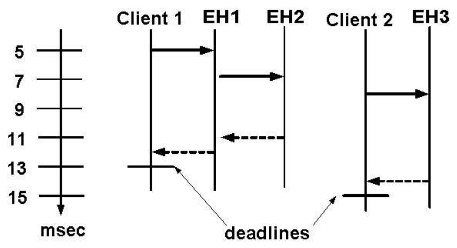

Figure 8. Timeline for Example 2

The composed model in UPPAAL for this example is very similar to the one shown in Figure 5, except for the modified models of Reactor1 shown in Figure 9 and EH1 shown in Figure 10. In this example, the reactor waits for either the reply to come back from EH2, or for another request message for EH1 or EH3 to arrive from a client, and dispatches whichever event arrives first: meanwhile, EH1 waits on the reactor until the pending reply comes back from EH2 and is delivered to EH1. Because of the synchronous nature of the two-way (request-reply) calls made between Client 1, EH1 and EH2, and between Client 2 and EH3, if the request from Client 2 to EH3 arrives just after the call from the reactor to EH3, then EH3 must finish and send the reply back to Client 2 before the reactor can return its thread of execution to EH1. *Even if the reply from EH2 arrived just after the call to EH3 was initiated, its handling is blocked until EH3 completes.*

Checking this model for deadline misses as in Example 1 yields a trace with a call sequence where Client 2 sends a request to EH3 that arrives and is dispatched to EH3 just after the call from EH1 to EH2 is made, leading to a system state in which Client 1 misses its deadline. The trace confirms that the interleaving call sequence from Client2 to EH3 can cause a crucial blocking delay when EH1 is waiting for the reply from EH2.

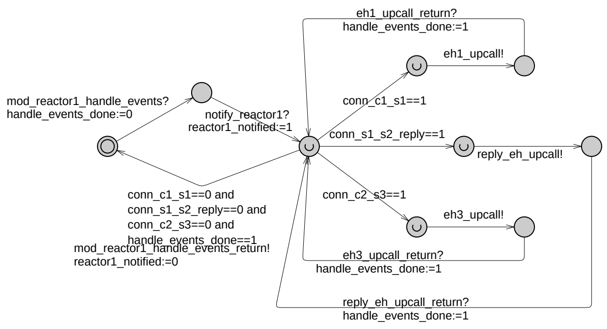

Figure 9. Reactor Model for Example 2

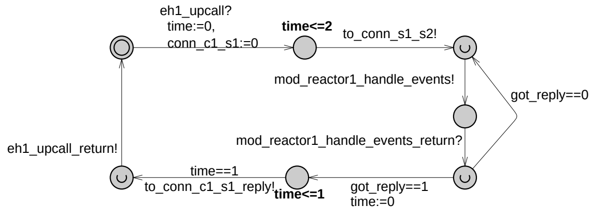

Figure 10. Example 2 - EH1 Waits on Reactor

Example 3 – multiple reactors, *WaitOnConnection* strategy: The problem raised in Example 2 by the WaitOnReactor strategy, in which nesting of calls by the reactor leads to a deadline miss in the 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 *WaitOnConnection*. The composed model is similar to Figure 5, except that as Figure 11 shows, the EH1 automaton models the fact that when EH1 sends a remote request to EH2 it waits directly on the connection for the reply, instead of waiting on the reactor as in Example 2.

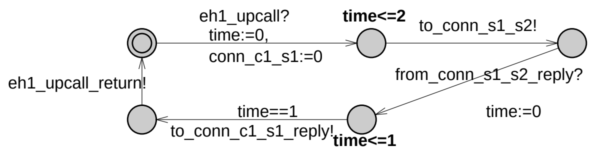

Figure 11. Example 3 - EH1 Waits on Connection

However, this approach in turn introduces further concurrency issues that must be evaluated, which our approach does through model checking. Consider for example another scenario based on the same deployment as in Example 2 (shown in Figure 7), but in which a call chain spans all three handlers, with EH1 depending on EH2 and EH2 depending on EH3. 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 when the single thread in Reactor1 is already in an upcall when there is an incoming request from EH2 to EH3. Figure 12 shows the timed call sequence. The deadline for Client1 at 15msec is missed when no progress can be made by the system after the request is sent from EH2 to EH3.

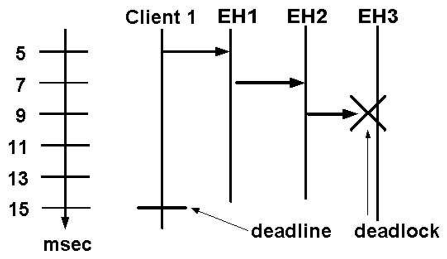

Figure 12. Timeline for Example 3

In real-time systems where each handler has a particular deadline, checking for a missed deadline as in the previous examples will reveal a deadlock indirectly, when the first deadline after the deadlock is reached passes without its completion criterion being satisfied. However, our approach also generalizes to systems where a deadlock will simply stop system progress, and will not produce an explicit failure such as a missed deadline. In those cases, checking the temporal logic query "A[] not deadlock" in the UPPAAL verifier will reveal any deadlock in the system.

Example 4 – multiple reactors, multiple threads: The scenario in Example 3 can be resolved by adding reactor threads, as illustrated in Figure 13. However, adding more threads does

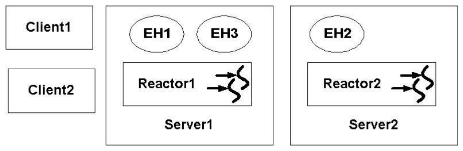

Figure 13. Deployment for Example 4

not guarantee deadlock freedom in general, since more than one client might call EH1 concurrently, again leading to deadlock as illustrated in Figure 14: any k threads in Reactor1 can be obtained by k distinct concurrent calls to EH1, leaving no threads to handle the call to EH3 and deadlocking each call chain.

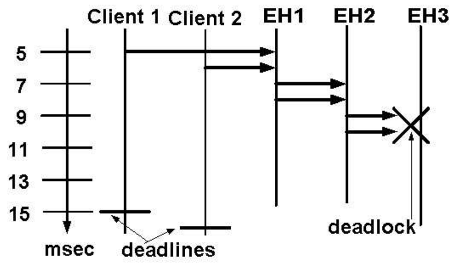

Figure 14. Timeline for Example 4

A more detailed analysis of this particular problem, and alternative protocols to avoid it, have been developed in complementary research [11], in which we examine alternative approaches to reduce the amount of analysis that must be performed through model checking. In that research, we have developed thread allocation protocols for deadlock avoidance by using the information about the call graph, *e.g.*, the depth at each position of each nested call chain.

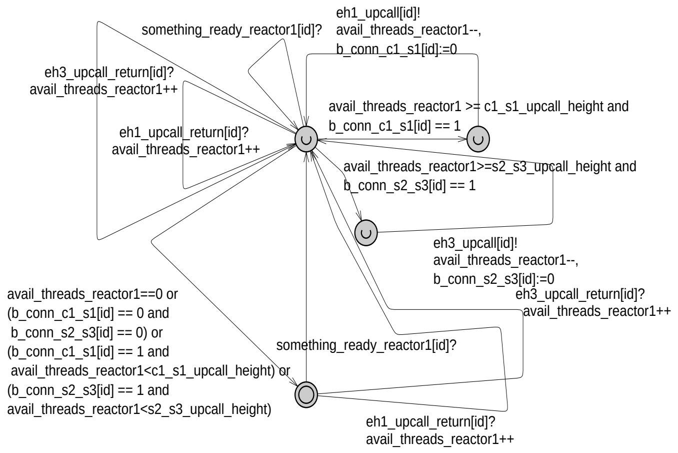
something_ready_reactor1[id]?
eh1_upcall[id]!
avail_threads_reactor1--,
b_conn_c1_s1[id]:=0
eh3_upcall_return[id]?
avail_threads_reactor1++
avail_threads_reactor1 >= c1_s1_upcall_height and
b_conn_c1_s1[id] == 1
eh1_upcall_return[id]?
avail_threads_reactor1++
avail_threads_reactor1>=s2_s3_upcall_height and
b_conn_s2_s3[id] == 1
eh3_upcall[id]!
avail_threads_reactor1--,
b_conn_s2_s3[id]:=0
eh3_upcall_return[id]?
avail_threads_reactor1++
avail_threads_reactor1==0 or
(b_conn_c1_s1[id] == 0 and
b_conn_s2_s3[id] == 0) or
(b_conn_c1_s1[id] == 1 and
avail_threads_reactor1<c1_s1_upcall_height) or
(b_conn_s2_s3[id] == 1 and
avail_threads_reactor1<s2_s3_upcall_height)
something_ready_reactor1[id]?
eh1_upcall_return[id]?
avail_threads_reactor1++

Figure 15. BASIC-P Reactor Model for Example 4

Figure 15 shows a reactor model that implements the BASIC-P protocol from that work [11]. We use a parameterized reactor model in which we can define the threadpool size. Whenever a request comes in to the reactor, the protocol is used to evaluate whether or not it is safe to allocate a thread to process the incoming request, based on information about the call graph previously provided to the protocol.

Using the complete model built using the multithreaded reactor model in Figure 15, we verified three things: (1) there is no deadlock with a single client and ≥ 2 threads in Reactor1; (2) without a deadlock avoidance protocol there is a deadlock when 2 clients try to make the same call sequence even if there are 2 threads in Reactor1; (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, Client2 could be blocked because Reactor1 reserves 2 threads for the call sequence initiated by Client1 to be able to complete.

Summary: The examples presented in this section motivate the need for detailed modeling of low-level middleware mechanisms, and evaluation of those models through model checking tools. Pursuing alternative formal techniques like the deadlock avoidance protocols described for Example 4 appears promising to reduce the state space that must be explored by model checking, and yet each such approach is likely to introduce further nuances that must be modeled. Therefore, these examples support our contention made in Section 2 that modeling and analysis should be done as an integral part of the system design and engineering process. As we note in Section 5, 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.

4 Related Work

In this section we describe related work in four main areas: model-integrated computing, model-driven middleware, customizable middleware, and fine-grain middleware building blocks. Our work presented in this paper complements and extends the related work in each of these areas.

Model-integrated computing: The Generic Modeling Environment [12] is a configurable toolkit for domain-specific modeling, which has been applied to middleware. Ptolemy II [13] is a modeling environment for embedded systems that provides a rich set of computation models. Our research complements these efforts, by providing fine-grained middleware mechanism models much as Giotto models for real-time embedded control systems [14] were integrated within Ptolemy II.

Model-driven middleware: CADENA [15] is an integrated GUI environment for building and modeling CORBA Component Model (CCM) [16] systems. The CoSMIC [17] toolset supports integrated model-driven component assembly, deployment and configuration. The low-level formal models we are developing, combined with the middleware mechanisms our models represent, can be integrated with these toolsets to provide fine-grained model checking and software synthesis capabilities over a common and reusable software base.

Customizable middleware: MicroQoSCORBA [18] reduces middleware footprint by *generating* customized instantiations of middleware for deeply embedded systems. Ubiquitous CORBA projects such as the CORBA specialization [19] of the minimal Universally Interoperable Core (UIC) [20] focus on metaprogramming of middleware. Zen [21] is a RT-Java [22] based real-time ORB and is also a highly customizable. Our work supplements these efforts with formal models of fine grained mechanisms that can be used, *e.g.*, in code generation.

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 [23] are similar to the reactor and event handler models we have already developed. Moreover, our approach also applies to other less similar environments, *e.g.*, to model and analyze the fundamental building blocks provided by TinyOS [24], which we plan to pursue as future work.

5 Conclusions and Future Work

Modeling and verification can play important roles in uncovering design errors from a very early stage in the development of distributed real-time systems. There is significant ongoing research that applies model-driven techniques to develop high-quality middleware. While current approaches for modeling middleware focus largely on easing the task of assembling, deploying and configuring middleware and middleware-based applications, a more formal basis for correct middleware construction and configuration in the context of individual applications is needed. Our approach, presented in Section 2, is designed to address that need.

The examples presented in Section 3 illustrate a variety of ways in which evaluating safety 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 to 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 the WaitOnConnection and WaitOnReactor; and (4) multiple additional on-line protocols, *e.g.*, for deadlock avoidance or security authorization. Furthermore, the constraints each application places on timing and other properties will alter the criteria by which system safety and liveness are evaluated.

The goal of our research is to address the difficulty of evaluating such a complex middleware environment, while preserving both rigor in analysis and tractability in applying our approach to real world systems. To meet that goal our future work will be to develop a growing set of robust, modular, and composable models of middleware building blocks, and integrating those models within model-integrated computing toolsets such as those described in Section 4. We will also continue our work on formally verified efficient protocols [11], along with the other optimizations described in Section 2 to reduce the burden of model checking.

There are a variety of model-checking tools that use different notations and formalisms. We have used the IF Toolset [25] and UPPAAL [10] to develop, simulate and verify timed automata models. We prototyped the models presented in Section 3 using the UPPAAL model checker for the purpose of illustrating a range of middleware models appropriate for different scenarios. However, in working toward our goal to develop a robust set of models that can model dynamic actions that are common in real-world middleware, the IF Toolset seems to fit our modeling requirements better than UPPAAL. UPPAAL uses a *rendezvous* model for communication between automata. The IF Toolset supports (1) dynamic composition capabilities that are useful for modeling run-time actions of the middleware like creating new event handlers and registering them with a reactor; (2) an asynchronous form of communication between automata that is well suited to modeling distributed middleware; (3) messages with parameters; (4) abstract data types; and (5) embedded C/C++ code integrated with the model.

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] G. J. Chaitin, “Register allocation & spilling via graph coloring,” in *SIGPLAN ’82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pp. 98–101, ACM Press, 1982.

[4] 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.

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

[6] 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 IEEE Real-Time and Embedded Technology and Applications Symposium*, (San Francisco, CA), pp. 190–199, Mar. 2005.

[7] C. Gill, V. Subramonian, J. Parsons, H.-M. Huang, S. Torri, D. Niehaus, and D. Stuart, “ORB Middleware Evolution for Networked Embedded Systems,” in *Proceedings of the 8th International Workshop on Object Oriented Real-time Dependable Systems (WORDS’03)*, (Guadalajara, Mexico), Jan. 2003.

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

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

[10] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on UPPAAL,” in *Formal Methods for the Design of Real-Time Systems*, no. 3185 in LNCS, pp. 200–236, Springer–Verlag, 2004.

[11] C. Sanchez, H. B. Sipma, V. Subramonian, C. Gill, and Z. Manna, “Thread allocation protocols for distributed real-time and embedded systems,” in *Submitted to FORTE 2005*, oct 2005.

[12] 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.

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

[14] 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.

[15] 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.

[16] N. Wang, D. C. Schmidt, and C. O’Ryan, “An Overview of the CORBA Component Model,” in *Component-Based Software Engineering* (G. Heineman and B. Councill, eds.), Reading, Massachusetts: Addison-Wesley, 2000.

[17] 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.

[18] A. D. McKinnon and D. Bakken and J. Shovic, “MicroQoSCORBA: A Reflective, QoS-Enabled, Configurable MicroCORBA With CASE Support,” in *Proceedings of the Second Workshop on Real-time and Embedded Distributed Object Computing*, OMG, June 2001.

[19] Manuel Roman and Roy H. Campbell and Fabio Kon, “Reflective Middleware: From Your Desk to Your Hand,” *IEEE Distributed Systems Online*, vol. 2, July 2001.

[20] Manuel Roman, “Ubicore: Universally Interoperable Core.” `www.ubi-core.com`.

[21] R. Klefstad, D. C. Schmidt, and C. O’Ryan, “Towards Highly Configurable Real-time Object Request Brokers,” in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2002.

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

[23] R. Hitchens, *Java NIO*. O’Reilly, 2002.

[24] 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.

[25] 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.