

Doctoral Topic Proposal: Towards Principled Fine-Grain Composition of Middleware

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

Wednesday, March 30, 2005

Abstract

Middleware for Distributed Real-time Embedded (DRE) systems has grown more and more complex in recent years due to the varying functional and temporal requirements of complex real-time applications. To enable DRE middleware to be configured and customized to meet the demands of different applications, a body of ongoing research has focused on applying model-driven development 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 formal basis for correct middleware composition and configuration in the context of individual applications is needed. While the modeling community has used high level formal models to uncover certain flaws in system design, a more formal, fundamental and lower-level set of models is needed to be able to uncover more subtle safety and timing errors introduced by interference between application computations, particularly in the face of alternative concurrency strategies in the middleware layer.

In this research, we will examine how formal models of lower-level middleware building blocks provide an appropriate level of abstraction both for 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 composing correct combinations of middleware mechanisms, and configuring those mechanisms for a particular application.

This research will thus have an impact on two different research communities. First, by modeling well-known low-level middleware abstractions such as reactors, acceptors, connectors, and service handlers, this work will establish concrete, canonical reference models for use by the formal systems modeling community. Second, by using those low-level models to evaluate particular concurrency and communication strategies in current use, this work will increase the rigor of current analysis and checking approaches used by higher-level model-based middleware development techniques.

1 Introduction

The complexity of middleware for DRE systems has grown in recent years due to the varying functional and temporal requirements of complex real-time applications. Development of flexible middleware that caters to the complex and varying QoS requirements for different applications is not trivial. Significant research has been done to make middleware more flexible and customizable through the usage of pattern-oriented techniques to develop middleware. Although this research has increased the flexibility and applicability of middleware to different kinds of applications, the choices for customization available to the application developer have increased and have become an increasing area of concern. Different combinations of middleware configurations result in different QoS properties for the application and therefore the selection of appropriate configurations is crucial for the correct functioning of each application.

To allow DRE middleware to be configured and customized to meet the demands of different applications, a body of ongoing research has focused on applying model-driven development 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 formal basis for correct middleware composition and configuration in the context of individual applications is needed. Formal models have been used traditionally by the modeling community to uncover flaws in system design early in the development lifecycle. But these high-level models, although formal, are not elaborated or refined throughout the entire development lifecycle. For example, decisions regarding the deployment topology, choice of middleware, *etc.* are not reflected in the high-level model. This may result in subtle safety and timing errors being introduced by *interference*, which we define in Section 2, between computations, particularly in the face of alternative concurrency strategies chosen at the middleware layer. Therefore a more formal, fundamental and lower-level set of models is needed to be able to uncover such errors.

In this research, we propose to use formal models of fundamental lower-level middleware building blocks to model a variety of kinds of middleware built from these building blocks. The resulting lower-level models can then be composed with high-level formal models of applications resulting in a *complete* model of a system. These high-fidelity models then can be validated for correctness using model checking techniques thus helping developers in composing correct combinations of middleware mechanisms, and configuring those mechanisms for a particular application. This approach has the following major impacts:

1. It brings more rigor in the model-based approach to middleware development pursued by the middleware community.
2. It brings more structure and fidelity in the form of canonical models of reusable and composable building blocks closely resembling the actual implementation, to the modeling community.

The rest of this proposal is structured as follows. Section 2 describes the current challenges faced in middleware composition, how current state-of-the-art in middleware research addresses these challenges and how the proposed solution supplements and advances the state-of-the-art.

In Section 3, we give an illustrative example of how *interference* may be caused by the choice of concurrency and communication strategies in the middleware layer, which in turn can affect safety and liveness properties of applications. Section 4 describes the proposed solution. Section 5 presents our research plan and finally Section 6 offers concluding remarks.

2 Background and Related Work

As middleware is applied to a wider range of complex distributed real-time systems, a fundamental tension between breadth of applicability and customization to the needs of each application becomes increasingly apparent. In this section, we discuss the challenges that face the middleware development community, how current state-of-the-art addresses these challenges and how our research supplements and advances the state-of-the-art in middleware composition. We begin by defining the *interference* in real-time systems.

2.1 Interference in Real-Time Systems

Interference occurs when the activities of multiple components in a system affect each other in ways that may produce adverse consequences for the system, for example when concurrency and synchronization semantics lead to deadlock or race conditions. We now give a more formal definition of *interference* in real-time systems.

Let $C = \{C_1, C_2, \dots, C_N\}$, $N > 0$ denote the 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 . Here resources represent any hardware or software entity a computation needs to use - threads, CPU, network, reactor (see Sidebar 1), *etc.* 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 at a particular instant in time. Note that $U(C_i, t) \subseteq R$, ($C_i \in C, t \in T$).

Two computations C_i and C_j ($i, j \leq N$ and $i \neq j$) *could interfere* with each other if $U(C_i, t) \cap U(C_j, t) \neq \emptyset$. In other words, if two computations require the usage of the *same* resource at the same time, then there is *interference* between them. Sometimes such *interference* could result in adverse consequences like deadline misses for computations that interfere with each other or introduce blocking factors into the execution times for computations.

As we show in Section 3, interference may occur because of DRE middleware configurations as well, where resources at the middleware layer, (*e.g.*, thread pools, reactor) are shared among different computations. We show a deadlock resulting from the sharing of a reactor and a thread between computations in one case, and in another case we show blocking delays introduced into execution times of computations.

The notion of *interference* is already prevalent in the field of programming languages, where an interference graph [1] is used by compilers for register allocation, the goal being that the total number of registers used should be as small as possible. The *interference* in this case is between program variables whose live-ranges [1] overlap. In this research, we aim to extend the notion of

interference to real-time systems. Just as interference graphs are used to analyze *interference* between variables in programming languages, we aim to use formal models to analyze *interference* between computations in real-time systems and how it impacts the real-time behavior of these computations. Specifically, as part of this research, we are interested in analyzing *interference* introduced by sharing of resources (*e.g.*, reactor, threadpools) at the middleware layer.

2.2 Challenges for Middleware Composition

Middleware for DRE systems must be designed to address the following challenges:

1. The middleware should provide common abstractions that can be re-used across different applications in the same domain.
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. To this end, middleware must be highly customizable.
3. The middleware should expose alternative mechanisms to resolve *interference* due to middleware components.
4. The application developer should be able to validate the correctness of the customized middleware in the context of the application.

While some of these challenges are addressed by previous research, a lot needs to be done to address others. For example, the first challenge has been addressed by research in the area of pattern oriented software development [2, 3]. Recent research on model-driven middleware addresses the rest of the challenges to a good extent, but a lot remains to be done still.

Reusable fine-grained abstractions: Frameworks like ACE [4] address the challenge of providing common domain-specific building blocks that could be used to build a range of middleware at different levels of abstraction. For example, the ACE framework provides building blocks like Reactor, Acceptor, Connector and Active Objects, for building DRE middleware - *e.g.*, for real-time scheduling (Kokyu [5]), distributed communication (TAO [6], nORB [7]), and component middleware (CIAO [8]). The presence of canonical fine-grain abstractions is not unique to middleware built on ACE. For example, in the sensor networks application domain, TinyOS [9] provides building blocks like Timer, ADC, RFM, Active Messages, *etc.* for building different kinds of sensor network middleware - *e.g.*, for reconfiguration, scheduling, group communication, and self-stabilization.

Customizable middleware: To enable customization of middleware according to application requirements, configuration capabilities must be provided. Currently this is typically done on an ad-hoc basis, based on knowledge scattered across expert developers, user manuals and newsgroups. Different combinations of configurations result in different functional and temporal

behavior and as a result such ad-hoc methods may not easily ensure system correctness. Significant current research is focused on applying Model Integrated Computing (MIC) [10] to the development of component middleware for distributed real-time embedded (DRE) systems [11]. Integrated toolsets are being developed to analyse the structural and semantic compatibility of application components. These tools also help the application developer in choosing suitable combinations of settings for the middleware configuration. Using this approach, a middleware expert can model the relationships between configuration settings and prevent the developer from choosing universally invalid sets of configuration settings. Even though this is an essential step in the right direction, we contend that there should be a more formal basis from which validity of configurations can be determined, so that configurations whose validity depends on contextual information can be evaluated as well.

Middleware interference issues: Developing complex DRE systems requires resolving *interference* among entities at different levels including middleware. We provide an illustrative example of *interference* in Section 3, where we show that it is important to consider this issue at fine granularity to develop *correct* DRE systems.

Abstraction level in modeling Distributed systems are deployed on a variety of platforms like CORBA, CCM, EJB, and TinyOS, each of which must be modeled. These models should be composable, reusable and verifiable so that they could be composed with high-level models of applications. This leads to the question of the the appropriate level of abstraction at which to model a platform. The models that we create should thus be at a level of abstraction that can support multiple kinds of middleware.

Validation of the complete system Finally, the complete system including the application and its middleware should be amenable to validation for correctness. The models of system infrastructure that we create should thus support composition with the application level models that we created so that a complete model of the system can be validated.

2.3 Related Work

There is significant ongoing research to address the above challenges for composing and configuring correct middleware. We describe the projects that are closest to this proposed model-driven middleware research.

2.3.1 Model Integrated Computing

Integration of embedded systems using different components require a great deal of *a priori* modeling and analysis. The key element in Model Integrated Computing (MIC) [10] is that it extends the scope and usage of models such that they form the “backbone” of a model-integrated system development process. The Generic Modeling Environment [12, 13] is a configurable toolkit for

creating domain-specific modeling and software synthesis environments. The generated domain-specific environment is then used to build domain models that are stored in a model database. These models are used to generate the applications or to synthesize input to different COTS analysis tools automatically. Ptolemy II [14] is another modeling environment for embedded systems that provides a rich set of computation models including the Giotto model [15] that provides an abstract infrastructure model for the implementation of embedded control systems with hard real-time constraints. Ptolemy II includes a code generator that generates E-machine code [15] from Giotto models.

Our research fits under the Model-driven Middleware (MDM) [16] paradigm which is the application of model-based techniques such as MIC to the domain of middleware. Moreover, our approach provides a more rigorous basis for middleware composition and validation 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, though that investigation is likely to be outside the scope of this dissertation. Unlike the Giotto model which creates a specialized concurrency environment for enforcement of timing properties, the proposed approach is to model canonical *existing* fine-grain middleware abstractions found in common use, as a basis for evaluation and composition of those elements.

2.3.2 Model-Driven Middleware Techniques

CADENA: CADENA [17] is an integrated GUI environment for building and modeling Corba Component Model (CCM) [18] systems. Its philosophy is based on the fact that reasoning about correctness properties is essential in component-based designs. Configuring a component is done through XML based descriptors which are tedious to write manually. Cadena provides a component assembly framework supporting a variety of visualization and programming tools for developing component connections. It provides model checking for verifying correctness properties of CCM systems derived from CCM IDL and XML. It does this based on specifications of a component along with component assembly information combined with Cadena specifications. It also provides facilities for defining component types, specifying dependency information and transition system semantics for these types. This proposed research is complementary to the CADENA approach providing fine-grain models for the middleware features and services on which the component environment is built.

CoSMIC: The CoSMIC [19] toolset provides an integrated component assembly, deployment and configuration environment for application developers, based on model-driven techniques. Using these tools, application developers can specify connections between components using a graphical interface. As part of this effort various modeling languages have been developed (PICML, OCML, BGML, etc. [16]) to assist application developers in building model-driven applications using component middleware. Model interpreters generate glue code, configurations and declarative component assembly information based on models of application components. Specifically, the Options Configuration Modeling Language(OCML) [20] allows a middleware developer to establish a rule base by which the application developer can choose a suitable set

of configuration options for the middleware infrastructure according to the application requirements. Another modeling language - Benchmark Generation Modeling Language (BGML) [21] forms the basis for automatically generating a test suite for instrumentation and measurement of application QoS properties.

The low-level formal models we will develop can be used to provide a *more exact* evaluation of safety and liveness properties that emerge from the composition of application and middleware features. This makes our approach both complementary to, and an improvement on, higher level modeling approaches that are simply based on feature sets and do not attempt to capture the more subtle structural and behavioral interactions that lead to *interference* from middleware. In the higher-level modeling approaches described in [21] and [22], they are able to conduct performance experiments for various combinations of features and configuration settings, which they can do using automated testing techniques to achieve coverage of feature/setting combinations, and thus analyze the best combinations of features and settings for a given application in a given context. This is very useful in practice, but does not explicitly model *why* the combinations lead to different performance results. The limitations of these approaches are thus that (1) that they must try all combinations of features and settings to be sure they've found the best ones and/or avoided the unsatisfactory ones; (2) even when they have mapped an application that way, the addition of a new feature or setting, or a change in the application, call graph etc., may have a significant impact on the previous profile of application performance due to the fundamental problem of *interference* examined in Section 3.

2.3.3 Customizable Middleware

MicroQoS CORBA [23] addresses the challenges of middleware footprint reduction by *generating* customized instantiations of middleware for deeply embedded systems. Using feature information about the underlying hardware, operating system abstractions, and middleware components, MicroQoS CORBA supports fine-grain configuration and composition of only the features needed for a particular application. While MicroQoS CORBA focuses on framework-independent generation of ORB infrastructure, related Ubiquitous CORBA projects such as LegORB [24] and the CORBA specialization [25] of the minimal Universally Interoperable Core (UIC) [26] focuses on a metaprogramming framework approach to DOC middleware. The Ubiquitous CORBA approach supports robust portability even across different DOC middleware paradigms *e.g.*, CORBA or SOAP [27]. It offers significant re-use of infrastructure, patterns, and techniques by generalizing features common to multiple DOC middleware paradigms and providing them within a minimal metaprogramming framework, thus also addressing the challenge of reducing middleware footprint. Zen [28, 29] is a RT-Java [30] based real-time ORB and is also a highly customizable. However, none of these research efforts offer a formal basis for composition. Our work supplements these efforts by providing formal models and implementations of fine grained mechanisms that could play a role in code generation.

2.3.4 Fine-Grain Middleware Building Blocks

We believe that the fundamental building blocks provided by TinyOS [9] are similarly suitable for fine-grain modeling like those in ACE, except that the domains that these two frameworks address are different - ACE provides building blocks for DRE applications and TinyOS provides building blocks for sensor network applications. However, we believe the modeling abstractions and ideas that we have proposed in this paper are applicable to a wide-variety of domains including sensor networks. For example, the principal force behind the design of Java NIO [31] is the Reactor design pattern. Our formal models of building blocks like Reactor thus have broad applicability, and our techniques for model development, composition and evaluation are even more generally applicable.

3 Example of Interference in DRE Systems

As we mentioned in Section 2.2, one of the key challenges to building *correct* DRE middleware is to resolve *interference* issues within and across layers of software. Modeling and verification techniques [32] can help to identify and resolve interference at an early stage in the system development lifecycle. Advances in model checking techniques have made this process more practicable in recent years. High-level formal models are very good tools during the initial stages of system specification, abstracting many details that are not relevant in that stage. However, during the system design and development stages, there tends to be a growing gap between the high-level formal models and the actual system implementation. Decisions are made, for example, as to the deployment topology of the application components across multiple processors and networks, and the middleware/OS platforms used. These decisions significantly impact the functioning of the system and bring into question the fidelity of the high-level model with respect to the actual implementation.

In this section, we provide an illustrative example that shows how two alternative compositions that are functionally equivalent have different safety and liveness properties because of *interference* issues in the application and middleware layers. One composition, while having predictable characteristics in terms of real-time properties, could result in deadlocks depending on the nature of the concurrency strategies used in the middleware and the nature of the application level dependency/call graph. The other composition, while not resulting in deadlocks, could result in reduced predictability due to execution blocking times. It is thus crucial to be able to analyze the properties of different compositions and to compare and contrast alternative compositions with respect to real-time, safety and liveness properties in the context of the application requirements and hosting environment constraints. To facilitate more faithful representation of a system, it is therefore necessary to model formally the software infrastructure on which the system is implemented so that alternative compositions can be compared with respect to safety, liveness and timing properties.

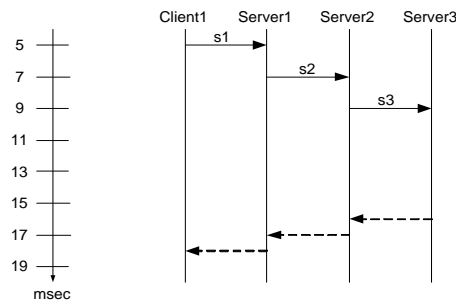


Figure 1: Timeline for Mode1

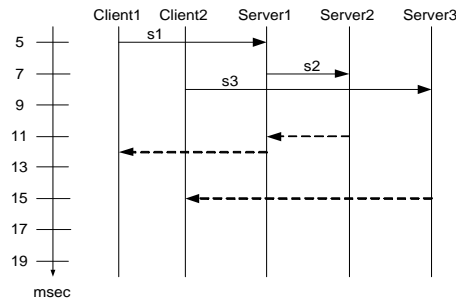


Figure 2: Timeline for Mode2

3.1 Motivating Example

The example application consists of two clients and three services and there are two modes of operation - Mode1 and Mode2. In Mode1, illustrated in Figure 1, at 5msec from the start of the application, Client1 calls Service1. After a computation that takes 2ms, Service2 calls Service3. Service3 does a computation that takes 7ms and then replies back to Service2. Service2 does a computation for 1ms before replying back to Service1. Service1 also does a computation for 1ms before replying back to Client1.

In Mode2, illustrated in Figure 2, the sequence of events is similar to Mode1, except that Service2 does not call Service3. Instead, it does a computation for 4ms and replies back to Service1. Moreover, Client2 makes a call to Service3 at 8ms after the start of the application (Note that Client2 does not run in Mode1).

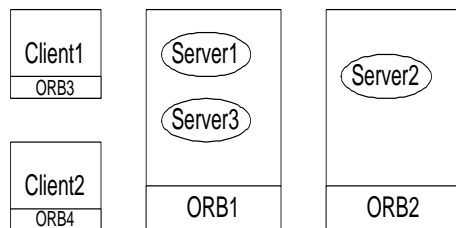


Figure 3: Deployment Topology

The example so far shows high-level models of interactions between concurrent system components. No decisions have been made so far regarding the middleware/OS platform or the

deployment topology. One can analytically verify that there is no timing violation with the above models. Once the interaction model has been checked, we now proceed to deploy the application. In this example, we choose the deployment topology as shown in Figure 3. Service1 and Service3 are hosted together and Service2 is hosted on another machine. We choose ORB middleware as the communication mechanism between the distributed components. Note that the decision to choose the deployment topology could be based on different factors including resource constraints or application requirements. We contend that a gap exists now between the high level model and the actual system, since the high level model does not consider the underlying infrastructure model, which as we describe next could result in problems because of *interference* issues in the middleware layer.

3.2 Interference from DOC middleware

Middleware typically offers different strategies to configure infrastructure mechanisms. Correct choice of strategies is crucial not only for the correct functioning of the infrastructure, but also is required to maintain safety and liveness properties of the application. Some combinations of infrastructure strategies could have adverse impacts on the functioning of the application. These are instances of the middleware infrastructure responsible for *interference* among computations in the application, possibly resulting in incorrect functioning of the application. In this section, we illustrate *interference* issues in the ORB middleware layer in the context of the above example. Although the discussion here is based on the ORB core implementations in TAO [6] and nORB [33], *interference* issues could occur in other implementations and other forms of middleware also.

CORBA [34] based ORBs are used in distributed systems with real-time constraints. Implementation of an ORB [6] involves mechanisms like Reactors and Leader-Follower thread pools (see Sidebar 1). To promise reuse of ORB middleware across a variety of domains and applications in those domains, ORB middleware often provides a wide set of configuration options so that it can be customized to suit the requirements of the application. In this section, we describe one such strategy used to configure the ORB core infrastructure – *Reply Wait Strategy* – to illustrate the importance of including this level of detail in a system model. In the course of this example, we show that the type of reply wait strategy chosen at one end-system affects the real-time characteristics as well as liveness properties of the application, and hence the reply wait strategy may contribute to *interference* in the system.

3.2.1 ORB Reply Wait Strategies

In CORBA, when a client makes a remote two-way function call, the caller’s thread needs to wait until it receives a reply back from the server before continuing to execute the calling method. This is in accordance with the semantics of a two-way function call. There are different strategies to wait for the reply, each having different implications for safety and liveness. Two different strategies to wait for the reply are illustrated here:

- Wait on Connection - the thread that sends the request waits directly on the connection for the reply

- Wait on Reactor - the thread that sends the request waits on the connection, using the ORB core reactor, for the reply.

We now illustrate the impact of these strategies on the safety and liveness properties of the example distributed application. In ORB literature, this kind of sequence of calls is termed “Nested Upcalls”. Without loss of generality, we first assume that there is a single thread of execution in the servers.

Sidebar 1: Key Patterns in TAO and nORB

The architecture of TAO and nORB is based on the network programming patterns described in [3]. We outline two fundamental patterns used in TAO and nORB that are relevant to the discussion in this proposal:

- **Reactor** is an event handling design pattern used in network programming to demultiplex events from multiple sources, possibly using just a single thread. This design pattern is used in ORBs to demultiplex and dispatch incoming requests and replies from peer ORBs. Event handlers like request and reply handlers are registered with a reactor. The reactor uses a synchronous event demultiplexer, *e.g.*, the UNIX *select* system call, to wait for data to arrive from one or more ORBs. When data arrives, the synchronous event demultiplexer notifies the reactor, which then dispatches the appropriate registered event handler based on the event source.
- **Leader/Followers** is an architectural design pattern that provides an efficient concurrency model where multiple threads take turns detecting, demultiplexing, dispatching, and processing requests and replies from peer ORBs.

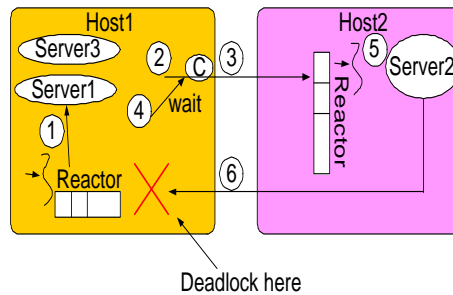


Figure 4: With a single thread, Wait On Connection strategy can lead to deadlock

Wait on Connection: In this strategy, illustrated in Figure 4, the following sequence of events takes place within the ORB layer:

1. As the remote call from Client1 is processed by ORB1, it makes an upcall to the servant implementation for Server1. In the subsequent discussion, we assume that the upcall is made in the same thread as the I/O thread that was listening on connections for remote calls.

2. As part its implementation, Server1 makes a remote call to Server2. Internally, ORB1 actively establishes a connection C to ORB2.
3. The parameters to the remote call are marshalled, a GIOP Request is formed and sent to ORB2 using C.
4. The sole I/O thread (that is also the upcall thread) in ORB1 waits for the reply on the connection C using a blocking `recv` call.
5. The request is received by ORB2 and dispatched to the skeleton code for Server2. Server2 skeleton code marshals the parameters and the *upcall* is made to the servant.
6. The servant implementation for Server2 calls Server3. Internally ORB2 tries to make a connection establishment to ORB1 so that it could send this request.

Since the sole I/O thread in ORB1 is blocked on a system call waiting for a reply from Server2, there is no thread to accept the incoming request. This results in a deadlock, where the ORB1 thread is waiting for a reply from ORB2 and the ORB2 thread is waiting for a reply from ORB1. Note that this situation occurs *only* because of interference issues in middleware and *not* because of any conditions occurring in the OS layer. The situation can be improved by having a pool of threads listening for input requests using the Leader-Follower model (see Sidebar 1). But even with this model, when the number of outstanding requests exceed the number of threads, the ORB ceases to accept any more requests and this can result in a deadlock in the case of nested upcalls.

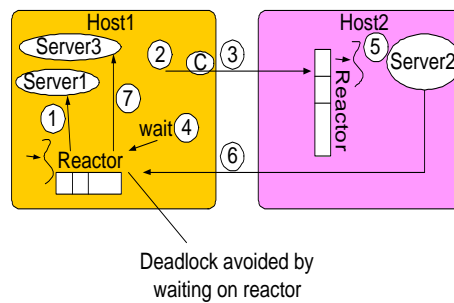


Figure 5: With a single thread, Wait On Reactor strategy prevents deadlock

Wait on Reactor: In this strategy, the sequence of calls is the same as the previous strategy until the request is written to the connection stream. After that, instead of waiting on the connection for the reply, the caller thread waits on the ORB core reactor, which provides synchronous demultiplexing of I/O events. This demultiplexing allows incoming requests to be accepted while waiting for replies (see Sidebar 1). The (nested) callback request from Server2 is accepted (7) and the call is completed eventually, thus avoiding deadlock (see Figure 5).

But this strategy introduces another interference issue in the form of *blocking delays*. It should be noted that the upcall for the incoming request is made in the same thread context as that of the outgoing call. There could be multiple incoming requests before the reply for the initial outgoing call arrives. The processing of the reply for the initial outgoing call can be done

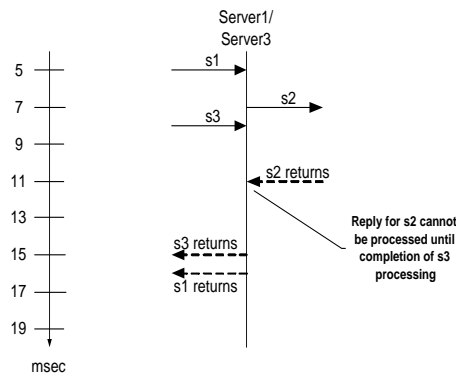


Figure 6: Timeline for Mode2 with interference

only after processing of all the incoming requests that arrived before its reply, is completed. This results in blocking delays in completion of outgoing remote calls. This is illustrated in Figure 6 in the context of the example application running in Mode2. The timeline shows the sequence of events that occur at ORB2. When the reply from Server2 arrives at 11ms, there is no thread to process that reply, since the upcall thread is already servicing the request from Client2 to Service3. The reply from Service2 is processed only after the reply from Service3 is sent to Client1. This introduces a blocking factor of 4ms in the processing of the reply from Service2 resulting in a delayed reply to Client2, which violates the specified property of the application that there is a timing violation if the reply to Client1 comes after 12ms. The new timeline with this *interference* is shown in Figure 7.

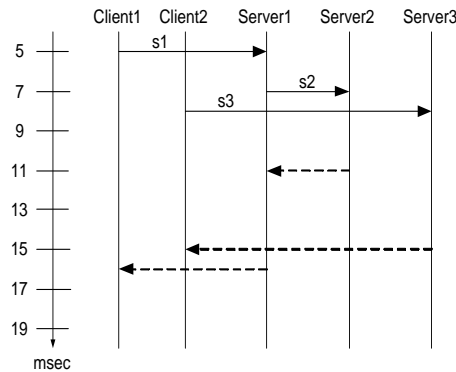


Figure 7: Timeline for Mode2 with interference

The above example illustrates two typical kinds of *interference* issues encountered in DRE middleware and reinforces the need to consider such issues when modeling real-time systems. It is therefore important to choose appropriate strategies at fine levels of detail in a middleware infrastructure. Depending on the nature of application properties, such as nested upcalls, this

choice may affect liveness and timing properties as shown in the example. Therefore, such details need to be taken into consideration to give a more faithful analysis of system models.

4 Resolving Middleware Composition Challenges - A formal approach

From the example in Section 3, it is clear that it is important to take into account *interference* issues arising at the middleware layer. We take the position that these issues can be resolved by taking a *principled and formal* model-driven approach to composition of fine-grain DRE middleware *building blocks*. Furthermore, 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. These formal models impact the development of middleware in the following ways:

- Forming more complete and detailed models of systems for verification by composing application models with infrastructure models, thus making the complete system model more faithful to the actual system
- Verification of safety, liveness and timing properties
- Documentation of collective engineering expertise in building DRE middleware
- Correct composition of fine-grain middleware elements enabling correct construction of systems

4.1 High fidelity Modeling of DRE Middleware

One of the challenges discussed in Section 2.2 is to determine the appropriate level of abstraction at which to model system software. To answer this question, one must look at the practical side of building systems and see the kinds of abstractions that system implementors use. For system developers, middleware provides a level of abstraction that is needed for portability and reusability and hence seems a good candidate for formal modeling. We contend however that lower-level building blocks (*e.g.*, Reactor, Acceptor, *etc.* provided by the ACE [4] framework for distributed object middleware) provide a more appropriate target for formal modeling. This gives us a basis on which we can build models of a variety of DRE middleware, built using these building blocks, by composing models of the building blocks. Furthermore these lower-level models express rather than hide crucial details related to concurrency and timing properties.

Our previous experience with system software composition resulted in a special-purpose distributed object communication middleware called nORB [7] with support for real time operation dispatching in the context of memory constrained networked embedded systems. We took a bottom-up compositional approach to the development of nORB [7], starting with lower level elements in the ACE framework and re-using as much as possible from it. We used building blocks like Reactor, Acceptor, Connector, CDR Stream, *etc.* from ACE [4]. Along with a bottom-up

approach for composition of middleware, the application itself was used as a guide for making tradeoffs between feature richness and footprint. For example, we used the application to guide the data-types to be supported and the format of messages to be exchanged. That work has given us insights into the application driven composition and customization of DRE middleware for this and other domains, resulting in composable models with a high degree of fidelity to how DRE middleware is built in practice.

Since nORB is built from basic building blocks, we believe that it serves as a good testbed for experimenting with the validity of our models of basic building blocks. We are currently developing a model of nORB using models of building blocks from ACE. We contend that the process of developing high-fidelity models is an iterative one - findings from the modeling process should aid in developing “correct” middleware and already existing middleware artifacts can be used in developing necessary abstractions for modeling. We are developing formal models of the different parts of nORB like its I/O subsystem, service handlers, marshalling/unmarshalling and request dispatching subsystems. The resulting complete model of nORB will be composed with sample application models and will allow key application properties to be verified using model-checking tools.

4.2 Tools and Formalisms for Modeling

Sidebar 2: Timed Automata

Timed automata [35] are finite-state machines equipped with sets of variables, called clocks, that measure the time elapsed between events. A timed automaton models the behavior of a single process or component of the system. The locations (nodes) of the automaton correspond to the different control points (states) of the process. A transition from one location to another corresponds to the execution of a statement. Timing constraints such as propagation delays, execution times and response times, are expressed as predicates on the values of the clocks.

A complex real-time system made up of M cooperating and communicating processes is modeled as the parallel composition of the corresponding M timed automata. To model inter-process communication, the transitions of the timed automata are labeled with input/output actions. Timed Automata thus provide a powerful formalism for modeling both functional and timing aspects of a system. Moreover, it allows one to model *relative* order and timing of events, thus allowing representation of both synchronous and asynchronous events.

Currently we are investigating the applicability of Timed Automata (see Sidebar 2) to model fine-grained composable DRE middleware mechanisms. We use a model-checking [32] approach to verify safety and liveness properties. There are a variety of model-checking tools that use different notations and formalisms. We are currently using IF-toolkit [36] and UPPAAL [37] to develop, simulate and verify timed automata models. Based on our preliminary work, IF-toolkit seems to fit our modeling requirements better than UPPAAL. UPPAAL uses the *rendezvous* model for communication between automata whereas IF-toolkit provides an asynchronous model

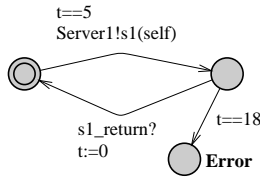


Figure 8a: Client1 automaton

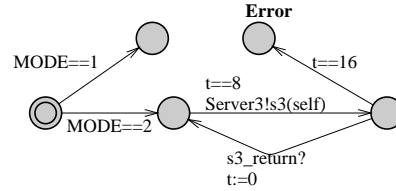


Figure 8b: Client2 automaton

Figure 8: Example to illustrate interference - Client Automata

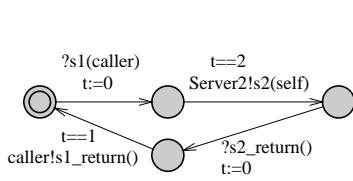


Figure 9a: Server1 automaton

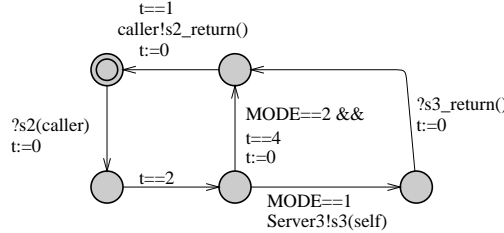


Figure 9b: Server2 automaton

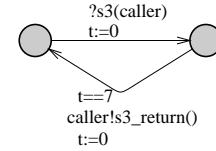


Figure 9c: Server3 automaton

Figure 9: Example to illustrate interference - Server Automata

of communication that is more suited to modeling distributed middleware. Moreover, for our purposes, IF-toolkit offers more modeling features than UPPAAL - *e.g.*, messages with parameters and abstract data types, and embedded C/C++ code.

A formalization of the example application discussed in Section 3.1 using timed automata is shown in Figures 8 and 9. We added some safety checks also as part of the model - Client1 times out after waiting for 18msec for the reply from Server1 and Client2 times out after waiting for 16msec for the reply from Server3. These timeouts are used in the models to detect timing violations in the system which could inturn result in safety or liveness violations. In this example, we are checking for a deadlock in Client1 by timing out after 18msec, if there is no reply from Server1. After the timeout, Client1 enters an *error* state. We are also checking for a timing violation in Client2. If Client2 does not get a reply from Server3 in Mode2, then this is considered a timing violation and Client2 enters an *error* state. Both these properties can be checked using a model-checker by expressing the above properties using temporal logic. For example, a temporal logic expression in the UPPAAL model checker to verify whether there is a deadlock in the system looks like the following - “E<> Client1.error”. This is a temporal logic expression which means “Is there any state along any state transition path where Client1 automaton is in *error* state?”.

The models of middleware building blocks that we will build will be composed with these high-level models to provide a high-fidelity model of the application. For example, a very simplified version of the formal model for a reactor and event handler is shown in Figure 10a and Figure 10b respectively. It should be noted that these models are far from complete and we will

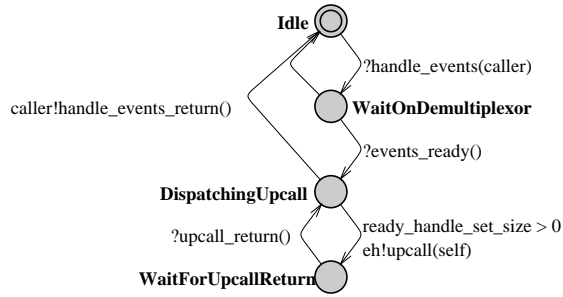


Figure 10a: Simplified formal model of a Reactor

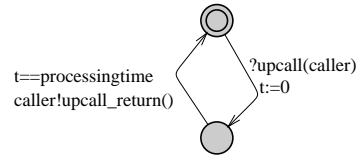


Figure 10b: Simplified formal model of an Event Handler

Figure 10: Simplified formal models of lower-level building blocks

enhance these models and create new models as we progress in our research.

Timed Automata are low-level formalisms and it is crucial to use the right kind of techniques to map the domain specific abstractions to the abstractions provided by the formalism. For example, in IF-toolkit each automaton is represented as a *process*. An IF process is concurrent with all other IF processes in the system, which is a composition of IF processes. An object can be represented using an instance of an IF process. A method call from one object to another can be mapped as sending an IF *signal* from the caller process to the callee process. Since a method call executes under the same thread of control, a caller process has to block for the return signal from the callee process. If we use this mapping, then there are issues such as the semantics of multiple threads sharing the same object, possibly executing different methods in the object. Each IF process instance can be only in one state at any point in time, which is not true in the case of an object that is shared by multiple threads, since each thread has its own stack and the state of the method execution (local variables) is stored as part of that stack. A similar issue occurs when we try to map recursive method calls. A possible solution that we are considering is to model an object as an IF process and model its methods using IF processes that are managed by the IF process that models the object. Each method call is then modeled by dynamically instantiating an IF process that models the method call. These are open questions and we will consider these issues as we progress further. We are also examining existing research [38] in this area and will try to leverage that work.

5 Research Plan

The following are the major remaining steps in our research:

1. Creating models of building blocks in ACE. To begin with, we plan to create and refine models of Reactor, Acceptor, Connector and EventDemultiplexer.
2. Implementation of the two different reply-wait strategies in nORB.

3. Creating a complete model of nORB using the models developed above.
4. Validate the fidelity of the model by using it to check known interference issues like the ones in the example discussed before.

An approximate timeline for the different activities is as follows:

- Development of models (April - Jun 2005)
- nORB modifications (Jul 2005)
- Experimental evaluation (Aug - Oct 2005)
- Dissertation writing (Oct - Dec 2005)

The success of our research will be based on the ability of our formal models to *predict* key behavioral properties for a composition - timing, performance, deadlock (or lack thereof), *etc.* - based on analysis of the model itself *before* it's used, and then empirically assess that prediction. We intend to use a combination of techniques and results from three different areas of research:

- Modeling formalisms and model-checking;
- Semantics of executable object-oriented models; and
- Mapping of the above semantics into the modeling formalism

Research in the area of modeling formalisms will help us with developing our formal models and expressing verifiable properties of the application. Research in the semantics of executable object models help us in implementing *executable* models of basic building blocks like Reactor, Acceptor, *etc.* These research efforts will give us insights into the semantics of object interactions and help us in developing models based on existing artifacts like ACE. Finally, there is significant ongoing research on implementing the above semantics in terms of the modeling formalisms, which we also intend to leverage.

6 Concluding Remarks

Modeling and verification plays an important role in uncovering design errors at a very early stage in developing 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 on easing the task of assembling, deploying and configuring middleware and middleware-based applications, a more formal basis for correct middleware composition and configuration in the context of individual applications is needed. Moreover, though these high level models help in uncovering certain fundamental flaws in system design, these high-level models do not form a complete model of the system due to *interference* in the form of

safety and timing errors introduced by the lack of consideration of infrastructure details in high level models.

In this proposal, we contend that developing and applying lower-level models of the actual infrastructure is also crucial, and assert that infrastructure building blocks form the right abstraction for modeling a variety of system infrastructure built from these building blocks. These composable formal models can help developers in composing correct middleware, configuring the middleware for a particular application, and developing faithful models of middleware. These composable formal models of existing fundamental middleware abstractions and our analytical and empirical evaluation of the efficacy of those models will be the main contributions of this work.

References

- [1] 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.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*. New York: Wiley & Sons, 1996.
- [3] 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.
- [4] Institute for Software Integrated Systems, “The ADAPTIVE Communication Environment (ACE).” www.dre.vanderbilt.edu/ACE/, Vanderbilt University.
- [5] C. D. Gill, D. L. Levine, and D. C. Schmidt, “The Design and Performance of a Real-Time CORBA Scheduling Service,” *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, Mar. 2001.
- [6] Institute for Software Integrated Systems, “The ACE ORB (TAO).” www.dre.vanderbilt.edu/TAO/, Vanderbilt University.
- [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] Institute for Software Integrated Systems, “Component-Integrated ACE ORB (CIAO).” www.dre.vanderbilt.edu/CIAO/, Vanderbilt University.
- [9] 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.

- [10] J. Sztipanovits and G. Karsai, “Model-Integrated Computing,” *IEEE Computer*, vol. 30, pp. 110–112, Apr. 1997.
- [11] G. Trombetti, A. Gokhale, D. C. Schmidt, J. Hatcliff, G. Singh, and J. Greenwald, “A Integrated Model-driven Development Environment for Composing and Validating Distributed Real-time and Embedded Systems,” in *Model Driven Software Development- Volume II of Research and Practice in Software Engineering* (S. Beydeda, M. Book, and V. Gruhn, eds.), New York: Springer-Verlag, 2005.
- [12] G. Karsai, S. Neema, A. Bakay, A. Ledeczki, 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] S. Neema, T. Bapty, J. Gray, and A. Gokhale, “Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems,” in *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE’02)*, (Pittsburgh, PA), Oct. 2002.
- [14] J. Liu, X. Liu, and E. A. Lee, “Modeling Distributed Hybrid Systems in Ptolemy II,” in *Proceedings of the American Control Conference*, June 2001.
- [15] 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.
- [16] A. Gokhale, K. Balasubramanian, J. Balasubramanian, A. S. Krishna, G. T. Edwards, G. Deng, E. Turkay, J. Parsons, and D. C. Schmidt, “Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications,” *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2004.
- [17] 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.
- [18] 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.
- [19] 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 Embedded 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.

- [20] 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.
- [21] A. S. Krishna, E. Turkay, A. Gokhale, and D. C. Schmidt, "Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, (San Francisco, CA), Mar. 2005.
- [22] 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), Mar. 2005.
- [23] A. D. McKinnon and D. Bakken and J. Shovic, "MicroQoS CORBA: 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.
- [24] M. Roman, M. D. Mickunas, F. Kon, and R. H. Campbell, "LegORB and Ubiquitous CORBA," in *Reflective Middleware Workshop*, ACM/IFIP, Apr. 2000.
- [25] 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.
- [26] Manuel Roman, "UbiCore: Universally Interoperable Core." www.ubi-core.com/Documentation/Universally_Interoperable_Core/universal%ly_interoperable_core.html.
- [27] J. Snell and K. MacLeod, *Programming Web Applications with SOAP*. O'Reilly, 2001.
- [28] 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.
- [29] R. Klefstad, D. C. Schmidt, and C. O'Ryan, "The Design of a Real-time CORBA ORB using Real-time Java," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing*, IEEE, Apr. 2002.
- [30] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.
- [31] R. Hitchens, *Java NIO*. O'Reilly, 2002.
- [32] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 1999.
- [33] V. Subramonian, G. Xing, C. Gill, C. Lu, and R. Cytron, "Middleware specialization for memory-constrained networked embedded systems," in *Proceedings of 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

- [34] Object Management Group, *Real-Time CORBA Specification*, 1.1 ed., Aug. 2002.
- [35] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.
- [36] 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.
- [37] 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.
- [38] I. O. Iulian Ober, Susanne Graf, “Validating timed uml models by simulation and verification,” in *Int. Journal on Software Tools for Technology Transfer*, Springer-Verlag, 2004.