Report Number: WUCS-2006-23

2006-05-01

Timed Automata Models for Principled Composition of Middleware

Venkita Subramonian

Middleware for Distributed Real-time and 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 as-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....
Read complete abstract on page 2.

Timed Automata Models for Principled Composition of Middleware

Venkita Subramonian

Complete Abstract:

Middleware for Distributed Real-time and 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 as-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 application-level formal models that are more abstract to uncover certain flaws in system design, a more 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 have examined how detailed formal models of lower-level middle-ware 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 each particular application.

2006-23

Timed Automata Models for Principled Composition of Middleware, Doctoral Dissertation May 2006

Authors: Venkita Subramonian

Corresponding Author: venkita@ieee.org

Web Page: http://www.cse.wustl.edu/~venkita/mw_models/

Abstract: Middleware for Distributed Real-time and 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 application-level formal models that are more abstract to uncover certain flaws in system design, a more 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 have examined how detailed formal models of lower-level middleware building blocks

Notes:
This research was supported in part by NSF CAREER award CCF-0448562. The timed automata models in

Type of Report: Other

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

TIMED AUTOMATA MODELS FOR PRINCIPLED COMPOSITION OF MIDDLEWARE

by

Venkita Subramonian

Prepared under the direction of Professor Christopher Gill

A dissertation presented to the Henry Edwin Sever Graduate School of Washington University in partial fulfillment of the requirements for the degree of

DOCTOR OF SCIENCE

May 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

TIMED AUTOMATA MODELS FOR PRINCIPLED COMPOSITION OF MIDDLEWARE

by

Venkita Subramonian

ADVISOR: Professor Christopher Gill

May 2006

Saint Louis, Missouri

Middleware for Distributed Real-time and 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 application-level formal models that are more abstract to uncover certain flaws in system design, a more 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 have examined how detailed 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 each particular application.

Contents

List of Tables

List of Figures

Acknowledgments

First of all, I'd like to thank the lord of Guruvayoor who is the guiding light of my life. Thanks to my parents who taught me how to be a good human being. I am not sure how I would pay my wife back for her endless patience and support during the past five years during which we have seen several ups and downs in our lives. Radhika, thanks for being with me. Thanks to my son Keshav, who brought the utmost happiness in our lives when we needed it the most.

I am greatly indebted to my advisor Dr. Christopher Gill for giving me the opportunity to come to Washington University as a research staff member and do my PhD simultaneously. If not for his constant encouragement, patience and countless hours of discussions with him, I would not have surpassed this phase of my life. Thanks, Chris, for being a sincere friend and mentor. I thank my other committee members - Dr. Douglas Niehaus, Dr. Ron Cytron, Dr. Aaron Stump and Dr. Ron Indeck for their insightful questions and comments on my thesis. I thank Dr. Niehaus and his students Tejasvi Aswathanarayana, Hariharan Subramaniam, Noah Watkins and Andrew Boie for all their help and support with the DSKI/DSUI instrumentation toolset which was of immense help in my thesis. I would not be doing justice if I did not mention a special thanks to Tejasvi Aswathanarayana for acceding to my innumerable requests to reboot the testbeds at ITTC.

I thank Dr. Marius Bozga, Dr. Iulian Ober and Dr. Joseph Sifakis who helped me with the IF toolkit. I thank the UPPAAL newsgroup members who helped me with the UPPAAL toolkit. I would like to thank my collaborators Dr. Douglas Schmidt, Dr. Henny Sipma, Cesar Sanchez, Dr. Chenyang Lu, Dr. Kirby Keller, Dr. Douglas Stuart, Jeanna Gossett, Tom Corcoran and Gan Deng for their insightful discussions and suggestions regarding the projects on which we worked together.

I would like to thank Huang-Ming Huang for his support and especially for his numerous suggestions on my final defense presentation. Finally, I would like to thank my friends - Yuanfang Zhang, Xiaorui Wang, Terry Tidwell, Morgan Deters, Stephen Torri, Radu Handorean, Delvin Defoe, Nanbor Wang, Yamuna Krishnamurthy, Irfan Pyarali, Bala Natarajan, Jeff Parsons, Krishnakumar Balasubramaniam, Jaiganesh

Balasubramaniam, Angelo Corsaro, Joe Hoffert, Dennis Noll, Roopa Pundaleeka, Pawan Mandalkar, Praveen Krishnamurthy, Cheng Huang and Frank Hunleth.

Venkita Subramonian

Washington University in Saint Louis
May 2006

Chapter 1

Introduction

The following quote extracted from [2] gives a concise definition of modeling and design which is well applicable to the work in this dissertation.

> "*Modeling* is the act of representing a system or subsystem formally. A model might be mathematical, in which case it can be viewed as a set of assertions about properties of the system such as its functionality or physical dimensions. A model can also be constructive, in which case it defines a computational procedure that mimics a set of properties of the system. Constructive models are often used to describe behavior of a system in response to stimulus from outside the system. Constructive models are also called executable models.
>
> *Design* is the act of defining a system or subsystem. Usually this involves defining one or more models of the system and refining the models until the desired functionality is obtained within a set of constraints."

The research presented in this dissertation deals with the development of executable or constructive models of reusable middleware building blocks that are commonly used to implement distributed real-time and embedded (DRE) systems. These executable models of middleware building blocks, when used in conjunction with higher level application models, enables a faithful analysis of timing and liveness properties during system design.

1.1 Motivation

Significant research over the past decade has made middleware more customizable through the use of pattern-oriented software frameworks [51, 50]. Although this has made middleware solutions suitable for a wider range of applications, managing the resulting multiplicity of customization options has become an increasing concern. To allow middleware to be customized to meet the stringent demands of different distributed real-time and embedded (DRE) applications, recent research has focused on applying model-driven techniques to DRE middleware [6]. Although current model-driven middleware approaches *facilitate* the correct assembly, deployment and configuration of DRE applications and middleware, we argue in this dissertation that a more detailed and formal basis for reasoning about timing and liveness properties in a variety of different middleware configurations is both desirable and possible.

Application-specific formal models have been used to uncover high-level design flaws early in system development [76, 108]. However, such models are currently difficult to maintain adequately as the system's specification is refined successively throughout the system development life-cycle. For example, decisions regarding the deployment of application components onto endsystems, or the choice of middleware concurrency strategies, often are not reflected in these high-level models. This may result in subtle timing and liveness hazards due to unexpected side-effects from *interference* from the middleware policies and mechanisms used by a set of distributed computations.

Interference. In general, the kind interference in DRE systems that we study occurs when part of a computation shares resources with or otherwise interacts with another part of a computation and in doing so impedes or obstructs its required progress. The interaction could happen at different levels - application level (*e.g.,* monitor objects), middleware level (*e.g.,* reactor, threadpool), OS level (*e.g.,* kernel buffers, file system data structures), or hardware level (CPU, network cards, CPU registers). Interference, if not controlled properly can produce undesirable side-effects in an application. For example, a non-critical computation executing on a CPU could interfere with a critical computation requiring the same CPU resource and thus cause the critical computation to miss a deadline.

To control interference, the causes of interference at each layer must first be identified and analyzed and then appropriate steps must be taken to mitigate its effects. For example, schedulability analysis techniques like RMA [64, 58, 15, 62, 99] analyze the effects of interference among a set of periodic computations sharing a CPU. Register allocation techniques [16] in a compiler analyze the interference among variables that are live at the same time and hence cannot be allocated to the same CPU register. The focus of this research is to capture and analyze two specific forms of interference occurring because of sharing of reactors [91, 90] and thread pools in the middleware layer - (1) blocking delays at a reactor and (2) exhaustion of threads in a reactor thread pool. (1) occurs when a I/O event is being dispatched by a reactor to the appropriate event handler [91, 90] and another I/O event is waiting to be dispatched to its event handler by the same reactor. (2) occurs when all threads in a reactor thread pool are occupied processing events dispatched by that reactor and none of them are available further for the reactor to dispatch pending events. We provide a more formal definition of these forms of interference in Chapter 3.

Figure 1.1 illustrates how a system that is specified with only time-driven constraints in its high-level model may be refined into a time and event-driven system during its design and implementation phases. In the high-level model, a purely time-triggered request is sent from a service requestor to a service provider through a middleware-implemented remote function call. However, the implementation of this remote function call goes through multiple middleware, OS and network processing stacks, each of which likely contains event-driven system elements. For example, Figure 1.1 shows middleware based event demultiplexers on the sender and receiver side endsystems, which enable a single thread on each side to be used to demultiplex I/O events (*e.g.,* packet arrivals and transmissions) from and onto multiple interaction channels (*e.g.,* sockets and pipes). Even the interaction channels are likely to be event driven, for example when IP packets arrive and are moved from the network interface card into an application-accessible transport-layer buffer. This dissertation addresses the problem of modeling and analyzing interference arising from the semantically rich interactions between system software components of the middleware layer of the system software architecture, which we now illustrate using a concrete example in the context of an Object Request Broker (ORB) [75].

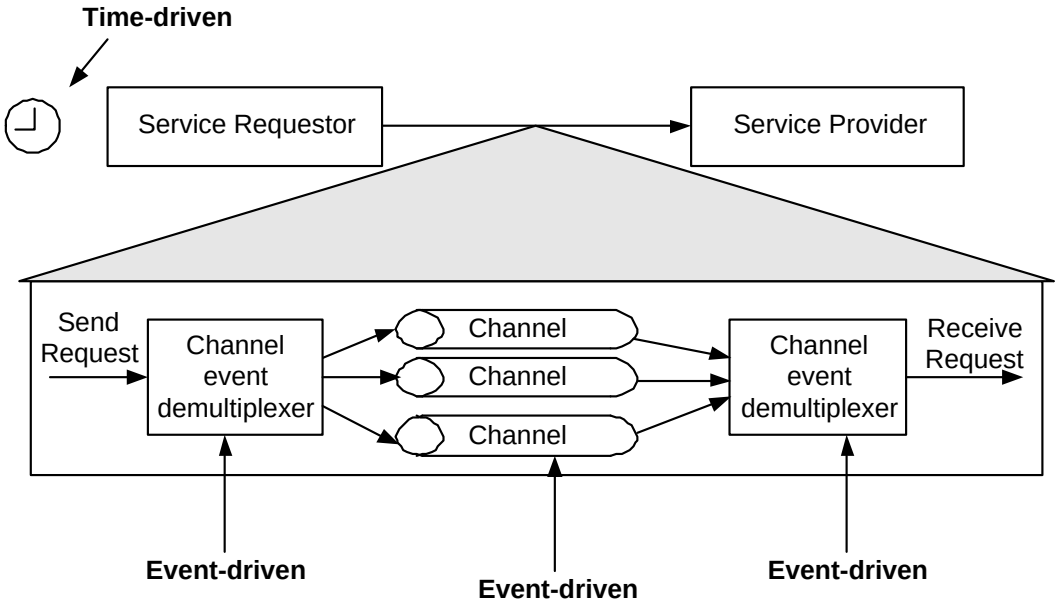


Figure 1.1: Remote Function Call as a Time and Event-driven Interaction

1.2 Interference example

Consider an example application that consists of two clients and three services with two modes of operation - Mode1 and Mode2. In Mode1, illustrated in Figure 1.2, at 5msec from the start of system execution, Client1 calls Service1. After performing a computation that takes 2ms, Service2 calls Service3. Service3 performs a computation that takes 7ms and then replies back to Service2. Service2 does a computation for 1ms before replying back to Service1. Service1 also performs a computation for 1ms before replying back to Client1.

In Mode2, illustrated in Figure 1.3, the sequence of events and actions 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 system execution (Note that Client2 does not run in Mode1).

The example so far shows high-level models of interactions between concurrent system components. No decisions have been shown so far regarding the middleware/OS platform or the deployment topology. In this example, we choose the deployment topology shown in Figure 1.4.

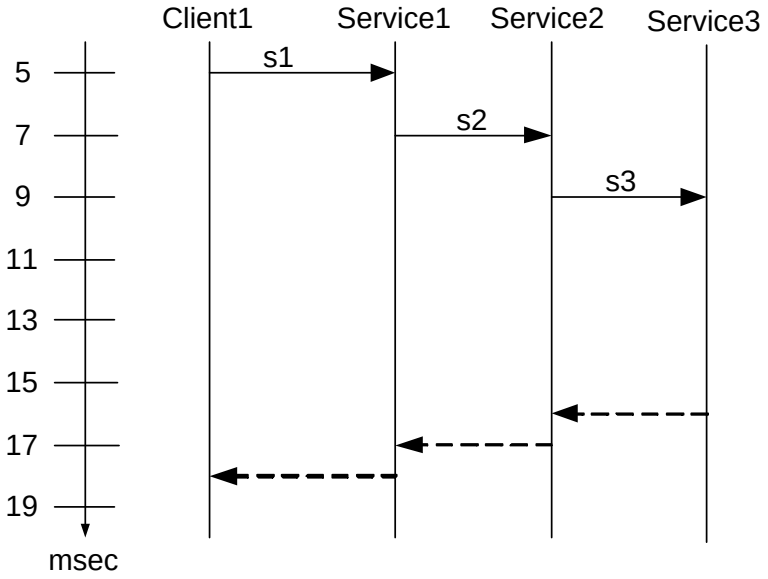


Figure 1.2: Timeline for Mode1

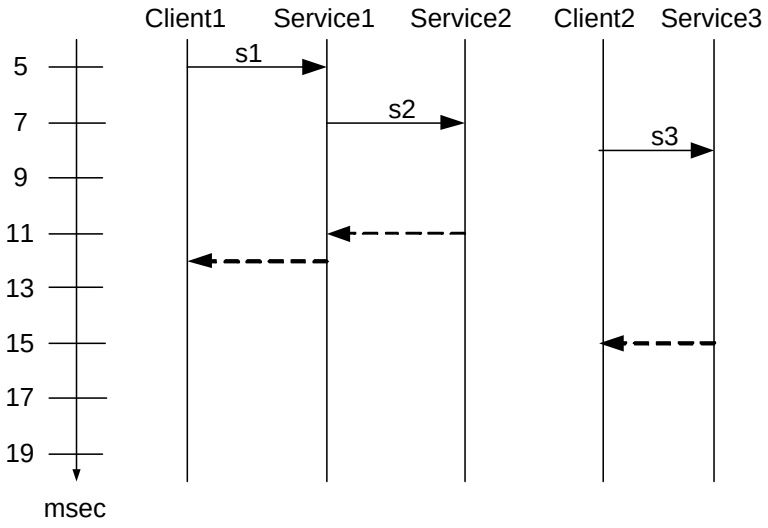


Figure 1.3: Timeline for Mode2

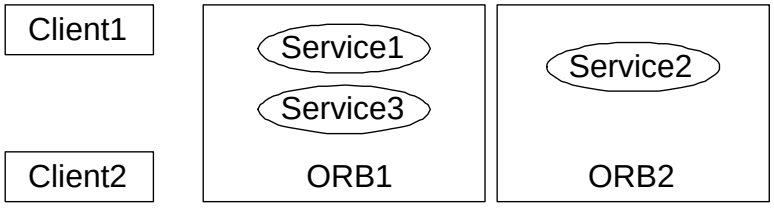


Figure 1.4: Deployment Topology

In this deployment topology, Service1 and Service3 are hosted together and Service2 is hosted on another machine. We choose ORB middleware (*e.g.,* TAO [50]) as the communication mechanism between the distributed components. Note that the decision to choose the deployment topology could be based on a variety of factors including resource constraints or application requirements. We contend that a gap exists 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 in the middleware layer.

1.2.1 Interference from DOC middleware

Middleware typically offers different strategies to configure infrastructure mechanisms. The correct choice of strategies is crucial not only for the functional behavior (delivering messages across sockets, waiting for replies, *etc.*) of the infrastructure, but also is required to maintain timing and liveness properties of the application. Some combinations of infrastructure strategies may have adverse impacts on the functioning of the application and yet current middleware modeling approaches do not consider the alternative configurations at a fine enough level of detail to be able to detect some forms of interference. In particular, the middleware infrastructure may itself introduce interference among computations in the application, possibly resulting in violation of application safety and liveness properties. In this section, we illustrate interference occurring in the ORB middleware layer in the context of a simple example that uses distributed objects communicating with each other to realize application-level goals. Although the discussion here is based in particular on configurations of middleware mechanisms seen in the ORB core implementations in TAO [50] and nORB [105], such *interference* could occur in other implementations and other forms of middleware as well.

CORBA [74] based ORBs are used in many distributed systems with real-time constraints. Implementation of an ORB [50] involves mechanisms like Reactors and Leader-Follower thread pools (see Sidebar 1). While modeling DRE systems, it is necessary to consider key infrastructure mechanisms like the ORB core reactor, the pool of threads used to receive incoming GIOP [21] requests, and the topology of

method invocations that generate outgoing GIOP requests. In this section we explain why this level of detail is important to verify correctness of the system with respect to timing and liveness properties. To support 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.

1.2.2 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. To motivate the need for modeling and analysis of middleware mechanism configurations, consider two different strategies used in TAO and nORB to allow a client to wait for the reply from a server.

- WaitOnConnection - the thread that sends the request waits directly on the connection for the reply
- WaitOnReactor - the thread that sends the request waits on a subsequent upcall from the ORB core reactor, when the reply has arrived.

We now illustrate the impact of these strategies on the timing and liveness properties of the example distributed application. In the 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 each of the servers.

Sidebar 1: Key Design Patterns in TAO and nORB

The architecture of TAO and nORB is based on the network programming patterns described in [97]. We outline the following fundamental patterns used in TAO and nORB that are relevant to the discussion in this dissertation:

- **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.
- The **Acceptor-Connector** design pattern decouples connection establishment between ORBs and request/reply processing in an ORB end-system once a connection is established. A Connector actively establishes a connection with a remote acceptor component and an Acceptor passively waits for connection requests from remote connectors, establishing a connection upon arrival of such a request, and initializing a service handler to process data exchanged on the connection.
- **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.

Wait on Connection. In this strategy, illustrated in Figure 1.5, the following sequence of events takes place within the ORB layer:

1. As the remote call from application component Client1 is processed by ORB1, it makes an upcall to the servant implementation for Service1. 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, Service1 makes a remote call to Service1. Internally, ORB1 actively establishes a connection C to ORB2.

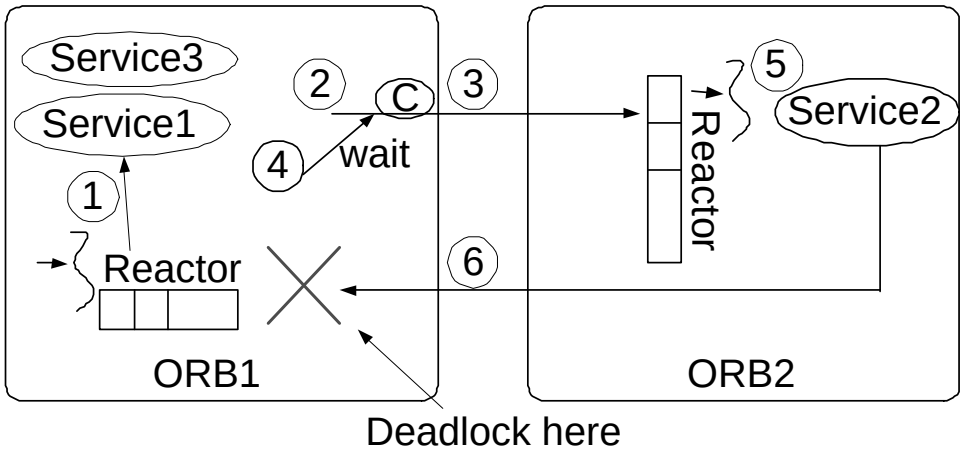


Figure 1.5: Deadlock with WaitOnConnection Strategy

3. The parameters to the remote call are marshaled, a GIOP Request is formed and sent to ORB2 using connection C.
4. The sole I/O thread (that is also the upcall thread) in ORB1 waits for the reply on connection C using a blocking `recv` call.
5. The request is received by ORB2 and dispatched to the skeleton code for Service2. Service2 skeleton code marshals the parameters and the *upcall* is made to the servant.
6. The servant implementation for Service2 calls Service3. Internally ORB2 tries to establish a connection 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 Service2, there is no thread to accept the incoming request. This results in a deadlock, where the ORB1 thread is waiting for a reply from Service2 and the ORB2 reactor thread executing Service2 is waiting for a reply from ORB1. Note that this situation occurs *only* because of interference in the 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 exceeds the number of threads, the ORB ceases to accept any more requests and this can result in a deadlock as well, as we show in Section 6.7 in Chapter 6.

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 Service2 is accepted (7) and the call is completed eventually, thus avoiding deadlock (see Figure 1.6).

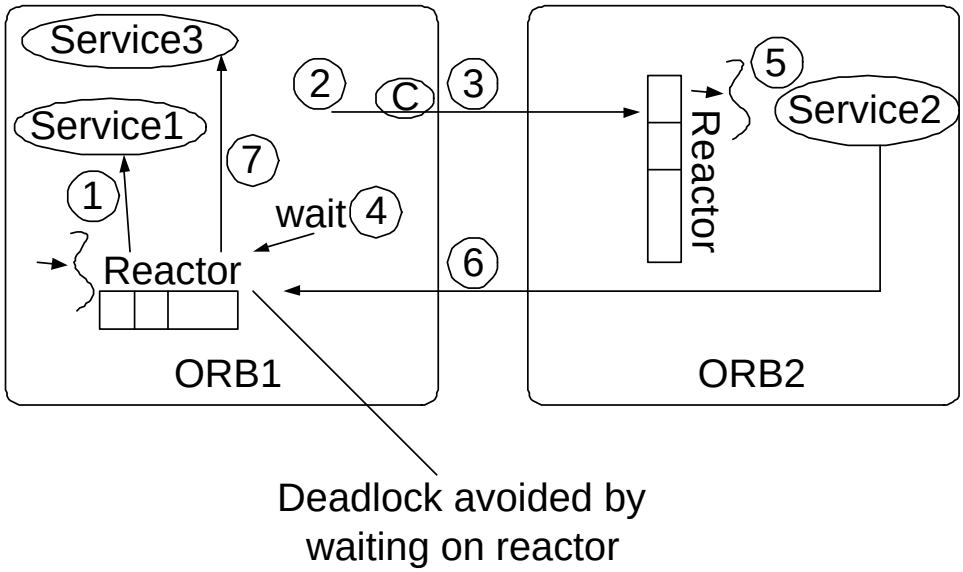


Figure 1.6: No Deadlock with WaitOnReactor strategy

However this strategy introduces another form of interference in terms 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 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 1.7 in the context of the example application running in Mode2.

The timeline in Figure 1.7 shows the sequence of events that occurs at ORB1. When the reply from Service2 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

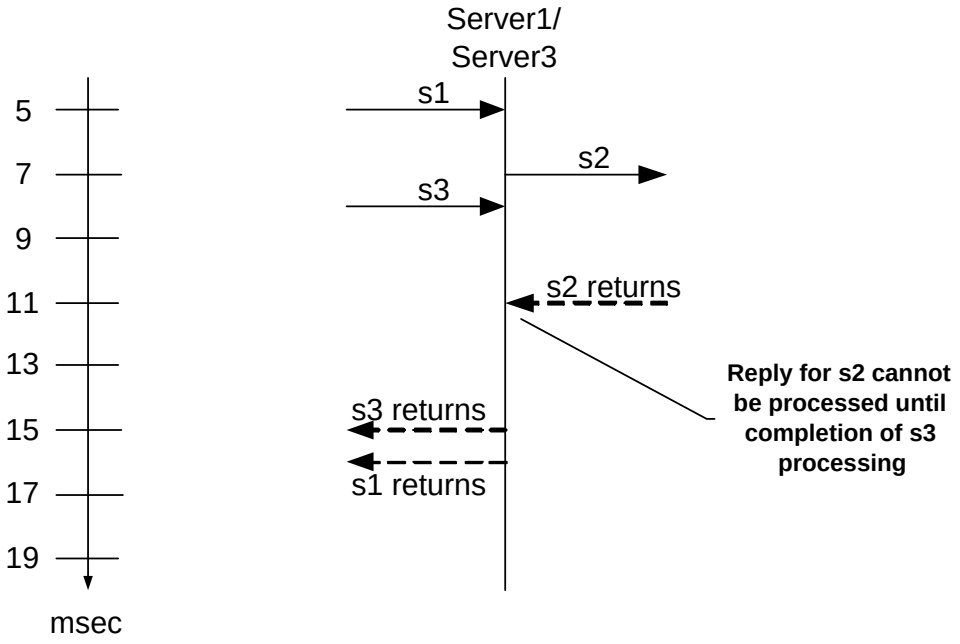


Figure 1.7: Cause of Blocking Delay - Local view at ORB1

Client2. This introduces a blocking factor of 4ms in the processing of the reply from Service2 resulting in a delayed reply to Client1, which may violate a timing property of the application. The new timeline illustrating the interaction between the different services is shown in Figure 1.8, which shows the effect of the blocking delay shown in Figure 1.7. Figure 1.8 shows that only after the reply from Service3 is sent to Client2, the pending reply from Service2 is processed and a reply sent to Client1.

The above example illustrates two typical kinds of interference issues encountered in DRE middleware - interference in the form of (1) deadlock caused by a combination of factors *e.g.,* WaitOnConnection reply wait strategy and a single reactor thread in ORB1 and (2) blocking delays caused by WaitOnReactor strategy. This example reinforces the need to consider such issues when modeling real-time systems. It is therefore important to choose appropriate strategies carefully and at fine levels of detail in a middleware infrastructure. Depending on application characteristics, such as use of nested upcalls, this choice may affect liveness and timing properties as was shown in this example. Therefore, such details need to be taken into consideration to give a more faithful analysis of system models.

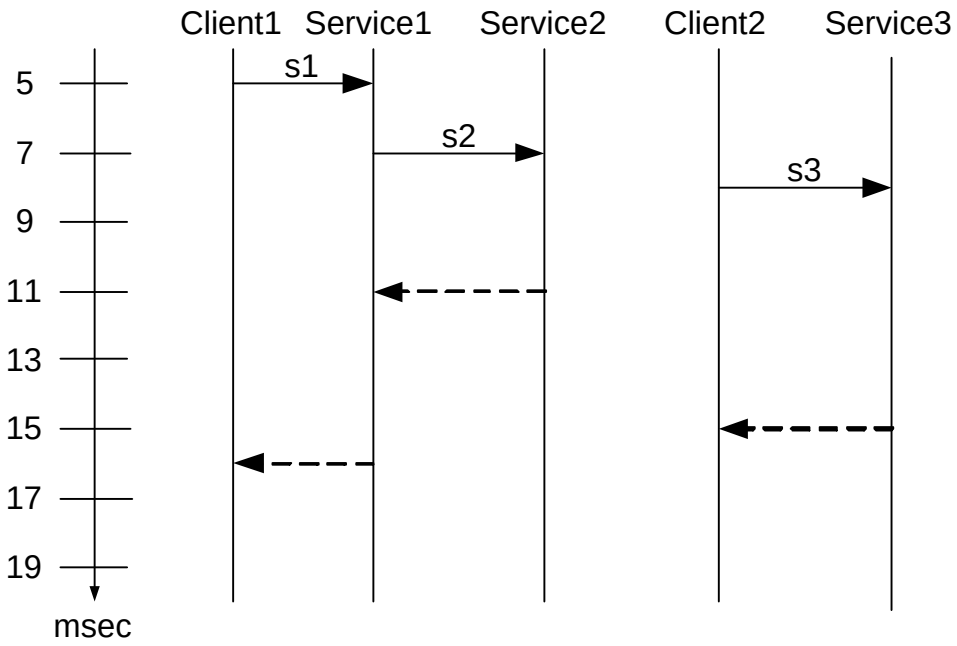


Figure 1.8: Effect of Blocking Delay - Global View

1.3 Challenges in Modeling DRE systems

The example in Section 1.2 illustrates the general concern that many of the abstractions used during high-level modeling, such as the notion of a purely time-driven or even a time-and-priority mediated interaction between the service requestor and the service provider, may become decreasingly representative of the system during its design and implementation. This may in turn result in a chasm between the high-level model and the actual implementation, *unless the abstractions used in the high level model can be refined during design and implementation.* Thus, a foundational set of formal models that can express both (1) high-level abstractions such as timed remote method invocations, and (2) low-level refinements such as concurrency and interaction semantics between the objects that implement the high-level model, is needed to support accurate verification of the high level model in terms of its low-level design and implementation.

Furthermore, the insights obtained from modeling and analysis should be made available and used while making design and development decisions, and vice-versa. Such a

close correspondence between the system modeling, analysis, design and development activities offers the following benefits: (1) more complete, detailed and executable models of systems, including their middleware infrastructure, can be composed and checked; (2) timing and liveness properties can be verified with greater precision; (3) a more rigorous and formal style of documentation can be used to capture and communicate detailed middleware engineering expertise that is currently represented less formally, *e.g.,* as *design patterns* [97]; (4) with more representative models and more powerful verification techniques, the extent to which systems must be "over-designed" can be reduced due to greater insight into the possible behaviors of the system.

Performing such verification at a realistic scale will require an approach that combines analysis using both static and dynamic models of the system. This involves the use of protocols [88, 89] that are provably correct using static analysis with respect to certain properties, *e.g.,* deadlock avoidance in systems with nested upcalls. As part of collaborative research with Dr. Henny Sipma, Cesar Sanchez and Dr. Zohar Manna from the Theory Group at Stanford University, we have investigated the static analysis approach [88, 89] as a complement to the executable models that were developed in the research presented in this dissertation.

The focus of the research presented in this dissertation is the development of dynamic or executable models of middleware and use model checking to verify the composition of middleware. These models enable us to model-check a variety of different middleware protocols including the deadlock avoidance protocols described in [88, 89]. In order to reduce the gap between high-level formal models and actual system implementation, this research addresses the key technical challenges outlined in Table 1.1.

1.4 Research Contributions

This dissertation makes the following major contributions to the state of the art in modeling DRE systems.

Table 1.1: **Challenges and Solution Techniques Presented in This Research**

Challenges	Solution Techniques	Effects and Results
Models of middleware should be at a sufficient level of abstraction such that they should be able to capture semantics of middleware building blocks and their interference effects such as the ones described in Section 1.2	A computational model and a modeling architecture based on timed automata, described in Chapter 3 in which the key models are of middleware building blocks that are reified in the ACE [51] framework.	The examples discussed in Chapters 6, 8 and 9 demonstrate that our models are capable of capturing a combination of concurrency semantics in middleware that includes the interference effects seen in Section 1.2.
Modeling of concurrent object middleware using a low-level formalism like automata is non-trivial - *e.g.,* modeling OS thread abstraction, mapping from an object model to a process model, state space explosion problem.	We present concrete engineering challenges and solutions for modeling concurrent object middleware using UPPAAL and IF in Chapters 4 and 5.	The empirical validation described in Sections 6.8.1 and 6.8.2 shows the effectiveness of our techniques in dealing with these challenges.
The models of middleware building blocks should be reusable and composable	We use communicating timed automata as our modeling formalism and tools that allow composition of these models. The level of abstraction of our models is similar to that of the reusable middleware building blocks in the ACE toolkit and hence our models can be reused to model most systems that are built using communication primitives in ACE as the middleware substrate.	The examples that we present in Chapters 6, 8 and 9 use a subset of middleware building blocks that are most commonly used in developing communication middleware. We modeled all these examples using those models.
The models should reflect the actual system closely and pinpoint any design flaws ahead of actual implementation	Our models are executable models that can be checked. We use model checking tools like IF and UPPAAL that generate detailed execution traces if there is a violation of the specified system requirements. We have developed tools and techniques described in Chapter 6 and 7 that further enable close correspondence between analysis of the model and the actual system.	The detailed execution and timing traces are used in the context of the various examples described in Chapters 6, 8 and 9 and have served as a valuable aid in debugging models, verifying hypotheses and uncovering design flaws.

It demonstrates the need for low-level middleware models. First, using illustrative example scenarios in the context of a distributed example using ORB middleware, it illustrates the need for including lower-level middleware details to adequately verify correctness of DRE systems middleware configurations.

It defines a reusable and relevant middleware modeling architecture. This dissertation defines a middleware-level modeling architecture to establish a common basis for developing concrete models of middleware using different modeling tools. To show the genericity of our proposed architecture, we realize this architecture using two different modeling tools that support timed automata - UPPAAL [7] and IF [12].

It provides executable models of middleware building blocks. This research develops executable timed automata [4] models of middleware building blocks in UPPAAL and IF, and demonstrates how liveness and timing analysis can be performed using model checking on system models that include the middleware infrastructure elements also. These timed automata models can then be used in conjunction with higher-level formal models to provide a faithful model of a system *including the middleware platform on which the system is deployed*, such that the composite models can be verified for correctness with higher fidelity to the system itself. In this research, we develop timed automata models of the following building blocks in ACE - select reactor, thread pool reactor with leader/followers, acceptor, connector, event handler, barrier synchronizer, and wrapper facades such as ACE_Pipe and ACE_SOCK_Stream for inter-process communication. We also demonstrate the modeling of the Half-sync Half-async and Active Object patterns, and the WaitOnReactor and WaitOnConnection reply wait strategies in the context of various examples described in Chapters 6, 8 and 9.

It provides new techniques for modeling middleware. This research also identifies key engineering challenges associated with building models of concurrent object middleware using the timed automata formalism in the context of UPPAAL [7] and IF [12] and also presents solutions to address these challenges. The following novel techniques are presented in this dissertation:

- Modeling objects and threads in the absence of native support for these abstractions in UPPAAL and IF.
- Modeling run-to-completion semantics in IF.
- Ordering rules and an idle catcher process to optimize the state space without over-constraining it, in IF.
- Workarounds for a limitation in IF to use procedures for waiting on condition satisfaction.
- Post-processing trace outputs from the IF model checker to produce object interaction traces and timeline traces that ease the process of model debugging and validation.

Though we have identified these challenges and devised solutions to them in the context of UPPAAL and IF, the solutions may find applicability in other modeling environments also.

It validates the effectiveness of this approach for realistic middleware configurations. We have demonstrated the effectiveness of the tools and techniques that we developed, in the context of various illustrative examples. Our research contributions include the modeling and implementation of a Deadlock Avoidance (DA) protocol [88, 89] that was invented as part of collaborative research conducted with Dr. Henny Sipma, Cesar Sanchez and Dr. Zohar Manna from the Theory Group at Stanford University. We implemented the BASIC-P DA protocol in the ACE TP Reactor, and also modeled and verified the protocol using our models. Our contributions also include modeling and verification of variants of the Gateway example that is available as part of ACE.

1.5 Dissertation Organization

This dissertation is organized as follows. Chapter 2 surveys related work in the area of modeling middleware. Chapter 3 presents a middleware-level system model

and a modeling architecture that forms the basis of our models. Realizations of this architecture using two different modeling environments - IF and UPPAAL - are presented in Chapters 4 and 5 respectively. Chapter 6 presents an application of our models by using simple but illustrative example scenarios and uses middleware domain expertise to validate the outcome of the analysis from the model execution. We also compare the effectiveness of different modeling techniques that we developed, in the context of these examples. Chapter 7 presents an evaluation of the fidelity of our models and compares the output from our models with that from actual execution. Chapter 8 presents a case study using our models in the verification of a deadlock avoidance protocol and Chapter 9 presents a case study using our models in the context of an application level gateway example. Chapter 10 presents concluding remarks about this research and its impact, and outlines future research directions.

Chapter 2

Survey of Related Work

We now survey related work from several perspectives of the modeling and middleware research communities. Integration of distributed embedded systems using different components, software as well as hardware, requires a great deal of *a priori* modeling and analysis followed by methodical implementation. Modeling enables the system designer to identify and analyze key design decisions that influence both functional and para-functional [22] aspects of a system. While the idea of using models for analysis and design of applications has been prevalent for some time, recently there has been significant ongoing research on applying model-based approaches to the middleware domain. Middleware is becoming an important and in many cases even an inevitable part of distributed real-time system implementations. More and more reusable services find their place in the middleware layer, contributing to the increased complexity of middleware and thus motivating the need for models of middleware building blocks that can be (re)used by the modeling community to model systems with a higher degree of fidelity.

Meanwhile from the perspective of the middleware community, some of the traditional software engineering principles like pattern-oriented [14, 97] development fall under the purview of good software engineering. These techniques are formal only to the extent that they enable a common vocabulary to communicate designs using standard notations like UML [111] and hence provide a means to understand designs. Although there is ongoing research in modeling middleware, a more rigorous and formal approach to middleware composition is needed therefore.

The work in this dissertation contributes to the application of formal model-based approaches to the middleware domain. In this chapter we survey four main areas of research that form the background for our research - (1) Model integrated computing (MIC) in DRE systems (2) Model-driven middleware (3) Formal techniques in concurrent and component based systems and (4) Middleware frameworks and execution environments.

2.1 Model Integrated Computing

Our research fits in to the broad research area of Model Integrated Computing [106, 72, 39, 42, 70]. The key idea in Model Integrated Computing (MIC) is to use models as a common basis throughout the system development process from analysis through design to implementation. Multiple views of a system are developed that help the designer to understand and analyze different aspects of the system under consideration, *before* committing to a particular implementation platform. In the case of distributed real-time and embedded systems, this would include not only the information processing aspects of a system, but also the physical architecture and the environment in which the system operates. These different models of a system are then integrated together to present a holistic view of the system and also to allow one to specify explicitly the dependencies and constraints among the various modeling views.

2.1.1 Applying MIC to DRE systems modeling

As part of the DARPA MoBIES and PCES projects, a modeling language called the Embedded Systems Modeling Language (ESML) [54] was developed and has been used to model component-based DRE systems that uses a publish-subscribe model for communication between the components. An example of a system in the avionics domain where ESML has been used is the Boeing Bold Stroke architecture [100]. Various tool chains are used to analyze the models either in isolation or in combination with other modeling views of the system. Some of the analysis tools that have been developed as part of the MoBIES and PCES research are AIRES [40], CADENA [44], Time Weaver - TimeWiz [107] and VEST [102]. A limitation of ESML used to be

that it mainly focused on the static structural aspects while largely ignoring the dynamic behavioral aspects of avionics mission computing software. This limitation is being addressed by recent work [17] on semantic anchoring which introduces facilities to attach behavioral semantics with models that use domain specific modeling languages like ESML. ESML models are used for high-level modeling of systems which do not take into account the interference effects of middleware elements, which is the focus area of our research. Whereas ESML provides models of higher-level building blocks (*e.g.,* processor, event channel, timer) that can be used to model DRE systems, our research provides models of lower-level middleware building blocks (*e.g.,* reactor, event handler).

Distributed Real-time Embedded Analysis Method (DREAM). DREAM [67, 66, 28] is an open-source tool and method that allows DRE system designers to do model-based schedulability analysis of time and event-driven DRE systems. DREAM offers a computational model called the DRE semantic domain [66]. The key elements in this computational model are tasks, timers, event channels and schedulers. Tasks are triggered either by a timer or external aperiodic events and tasks communicate among themselves by means of an event channel. Within this computational model, DREAM considers the problem of deciding the schedulability of a given set of tasks with time and event-driven interactions. By using timed automata models for each of the elements in the computational model, the schedulability problem is converted [28] into a reachability problem in the composed model using a model checking tool like UPPAAL. DREAM also provides a model transformation facility by which a model of the DRE system expressed using a domain specific modeling language (*e.g.,* ESML [54]), is transformed using model transformation [67] tools to timed automata models in the DRE semantic domain.

Even though our approach is similar to DREAM in that we use timed automata models to verify system properties, the problems that these two bodies of research address are different. Whereas DREAM addresses the problem of deciding schedulability of a set of tasks under the DRE semantic domain, our research addresses the problem of correct composition of middleware elements that are at a finer level of granularity than the elements in the computational model offered by DREAM. Both these kinds of analysis are important - while the higher level computational model provided by

DREAM helps the DRE systems designer to address the schedulability problem in time and event-driven systems, the lower level computational model that we describe in Chapter 3 helps the DRE system designer to choose a set of middleware configurations that is appropriate for the DRE application. Moreover, the computational model in DREAM makes an assumption that all communication between tasks use an event channel and the communication between tasks and event channels themselves are abstracted away using synchronized transitions in UPPAAL. During actual implementation, these synchronized transitions are likely to be realized using communication middleware like a CORBA ORB which, as we explained in Chapter 1, could have different configurations which impact the timing and liveness properties of a DRE system in different ways. Hence a more detailed model of the fundamental middleware elements (*e.g.,* reactors, event handlers, reply wait strategies) that constitute a middleware communication mechanism like an ORB or event channel is necessary, which is the focal point of the research in this dissertation.

GME. The Generic Modeling Environment [54, 70] is a configurable toolkit for creating domain-specific modeling and software synthesis environments. This toolkit uses meta-models to generate domain specific modeling languages and 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.

Although GME provides an environment for generating modeling environments, the semantics of the domain and the domain elements themselves must still be defined in the form of a computational model and associated semantics. As part of this research we propose a computational model for modeling interactions among middleware building blocks and define the semantics of this model in terms of communicating timed automata. We believe that the models developed in this research can become part of a middleware domain modeling environment like CoSMIC [37] that uses GME for generating the modeling environment.

Ptolemy. Ptolemy [65] is a framework for experimenting with heterogeneous models of computation. It is another modeling environment for embedded systems that

provides a rich set of computation models including the Giotto model [46] that provides an abstract infrastructure model for the implementation of embedded control systems with hard real-time constraints. Ptolemy includes a code generator that generates E-machine code [46] from Giotto models. Unlike the Giotto model which creates a specialized concurrency environment for enforcement of timing properties, the approach in this dissertation is to model canonical *existing* fine-grain middleware abstractions found in common use, as a basis for evaluation and composition of those elements.

Conceptually, Ptolemy supports the basic idea of the work in this dissertation that one model of computation may not be appropriate to analyze a system completely. While Ptolemy allows a system designer to compose different models of computation and analyze system properties, this research focuses on defining the behavioral semantics of *a* computation model in which the key computation elements are commonly used middleware building blocks. Thus it is conceivable that the computation model developed in this research and its semantics can be offered as a computation model in Ptolemy.

As future work we plan to investigate the suitability of integrating our formal models within the GME and Ptolemy environments, though that investigation is outside the scope of this dissertation.

2.2 Model-driven Middleware

Our research falls directly under the Model-driven Middleware (MDM) [36] paradigm, which is the application of model-based techniques such as MIC to the domain of middleware. Our approach provides a more rigorous basis for middleware composition and validation than current model-based middleware configuration techniques.

CoSMIC. The CoSMIC [37] 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 [6], OCML [108], BGML [59], etc.) 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) [108] 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) [59] forms the basis for automatically generating a test suite for instrumentation and measurement of application QoS properties.

The low-level formal models we have developed 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 [59] and [6], it is possible to conduct performance experiments for various combinations of features and configuration settings, which can be done 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) one 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 one has 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 1.2.

Model checking. Advances in model checking techniques have made state space exploration an attractive option for reasoning about system properties. Examples of model checking tools include SPIN [49], Bogor [86, 27], UPPAAL [7] and IF [11, 12] of

which UPPAAL and IF offer support for timed automata models. Several techniques have been developed by the model checking research community which make model checking more practicable. As part of our research, we developed middleware domain specific state space optimization techniques that complement the more general state space reduction techniques that are being developed by the model checking research community.

While the above tools use abstractions of actual systems expressed in some modeling language, tools like Verisoft [35, 79] systematically explore the state spaces of systems composed of several concurrent OS processes executing arbitrary C/C++ code. Verisoft implements a state-less search algorithm [35] to do a systematic exploration of the state space. The execution of the system processes is controlled by an external process called the *scheduler*. This process observes and controls execution of the system processes by suspending and resuming their execution. All sources of non-determinism are fully controlled by the scheduler process. Our research differs from the Verisoft approach in that it captures the interactions at a much more fine-grain level of abstraction (middleware building blocks) than OS processes.

A number of other formal models have been developed using model checking to reason about system properties. Specifically, [23] uses the Bogor model checker to model the behavior of a real-time event channel. [41] uses Finite State Processes(FSP) [52] and the associated Labeled Transition System Analyzer (LTSA) [1] tool to model check component-based real-time and embedded software. In contrast to the above work, the work in this dissertation develops formal models of reusable middleware building blocks that are more fundamental and reusable and hence can be used to model a variety of middleware services including middleware protocols, application gateways, and application-specific combinations of middleware building blocks.

2.3 Formal techniques in concurrent and component-based systems

Reasoning in concurrent and component-based systems. Task/Scheduler Logic (TSL) [83, 82] has been used to reason about concurrency in component based

software systems. Each component may come under the purview of a hierarchy of schedulers, each imposing its own set of restrictions on the type of resources that can be used. TSL uses first order logic to represent tasks, resources, locks and schedulers. Such reasoning is essential in component based systems to make more efficient uses of resources. Components are executed in environments which may be different from the environments in which they were developed. TSL can be used to find errors in system code, for example using a lock in a component which will eventually be run as an interrupt handler. There are different kinds of locks like regular mutex locks, recursive locks, readers-writer lock, *etc.* Based on the environment and the call graph of functions, TSL can be used to infer the type of lock to be used by a particular component under a particular context. The research in this dissertation develops executable models instead of the static analysis approach used in TSL and hence the model execution produces detailed traces of execution thus enabling us not only to reason about system properties but also to identify *why* system properties are satisfied or violated in terms of the semantics of the execution components involved.

Reasoning in CCM. Apart from providing an integrated environment for building and modeling CORBA Component Model (CCM) [110] systems, CADENA [44] also supports reasoning about correctness properties in component-based designs. CADENA provides model checking for verifying correctness properties of CCM systems derived from CCM IDL and XML. It does this based on behavioral 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. Our approach is similar to the CADENA approach in that both specify behavioral semantics of software, except that CADENA models capture higher level application component model behavior whereas the models developed in this research are lower-level middleware models of reusable building blocks that can be used to build a variety of middleware subsystems including CCM container implementations.

Formal techniques in CORBA. [55] introduces new stereotypes in UML and defines ways to map these stereotypes to the Finite State Processes(FSP) [52] process algebra and then do model checking for deadlocks. Apart from the single threaded

synchronous request scenario that is discussed in the paper resulting in a deadlock, we also consider scenarios which may not result in a deadlock based on the appropriate choice of strategies used to wait for a reply in the underlying infrastructure. Moreover, we also check the timing related properties of the system. Kamel's work [53] uses model checking to verify the GIOP protocol used in CORBA based systems. Duval's work [26] also uses a model checking approach to verifying CORBA based systems. [20] uses a formal language TRIO to specify CORBA-based distributed applications and uses a proof-based approach for reasoning about systems and verifying their correctness. Our work differs from the above work in that it provides executable formal models for *fundamental building blocks* that can be used to verify a variety of middleware services including CORBA implementations.

2.4 Middleware Frameworks and Execution Environments

E-machine. The Embedded Machine or "e-machine" [46] is a virtual machine that mediates in real time the interaction between software processes and physical processes. The e-machine runs E-code which is platform-independent and is generated by a compiler based on higher level specifications of the embedded system. The E-code specifies the timing of application tasks with respect to external events. The E-code can be checked for time safety [45] to determine platform specific inconsistencies. We believe the ACE toolkit can be used as a substrate to which target code generation, similarly to how E-code is used as a target language, based on which a variety of different applications can be developed: modeling the ACE substrate with timed models as we do, broadens the area of applicability.

Customizable Middleware. MicroQoSCORBA [3] 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, MicroQoSCORBA supports fine-grain configuration and composition of only the features

needed for a particular application. While MicroQoSCORBA focuses on framework-independent generation of ORB infrastructure, related Ubiquitous CORBA projects such as LegORB [87] and the CORBA specialization [69] of the minimal Universally Interoperable Core (UIC) [68] focuses on a meta-programming framework approach to middleware. The Ubiquitous CORBA approach supports robust portability even across different middleware paradigms *e.g.,* CORBA or SOAP [101]. It offers significant re-use of infrastructure, patterns, and techniques by generalizing features common to multiple middleware paradigms and providing them within a minimal meta-programming framework, thus also addressing the challenge of reducing middleware footprint. Zen [57, 56] is a RT-Java [8] based real-time ORB that is also highly customizable. However, none of the these research efforts offers a formal basis for composition with respect to timing and liveness properties. Our work complements these efforts by providing formal models and implementations of fine grained mechanisms they may leverage. For example, the principal force behind the design of Java NIO [48] 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.

CIAO. The Component Integrated ACE ORB (CIAO) [109] is a QoS-aware open source implementation of the Lightweight CORBA Component Model (CCM) [110, 73] specification. CIAO provides a component-oriented paradigm to distributed, real-time, embedded (DRE) system developers by abstracting DRE-critical systemic aspects such as QoS requirements as installable/configurable units supported by the component framework. The work in this dissertation can be used to model the behavior of parts of the run-time environment in CIAO, for example a CCM container.

TinyOS. The fundamental building blocks provided by TinyOS [47] 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 dissertation are applicable to a wide-variety of other domains including sensor networks.

OSEK VDX. OSEK VDX [78] is a set of interface specifications for operating systems, communication and network management in the automotive domain. The OSEK operating system is targeted to run on micro-controllers and therefore designed to require a minimum of hardware resources like CPU and memory. These specifications enable automotive OEM and third-party ECU (electronic control unit) suppliers to use a standardized set of APIs to facilitate system integration, thus making automotive applications more portable, reusable and interoperable. Models of the pattern-oriented building blocks in OSEK VDX can be developed similar to the models of fine-grain building blocks that we developed in this research. For example, a task in OSEK OS can wait on multiple events at a time using the *WaitEvent* OSEK function. This is similar to the Reactor pattern [97]. The Active Object [60] pattern can be used as in [85, 84] to separate the communication subsystem in OSEK VDX from the application.

Chapter 3

Middleware Modeling Overview

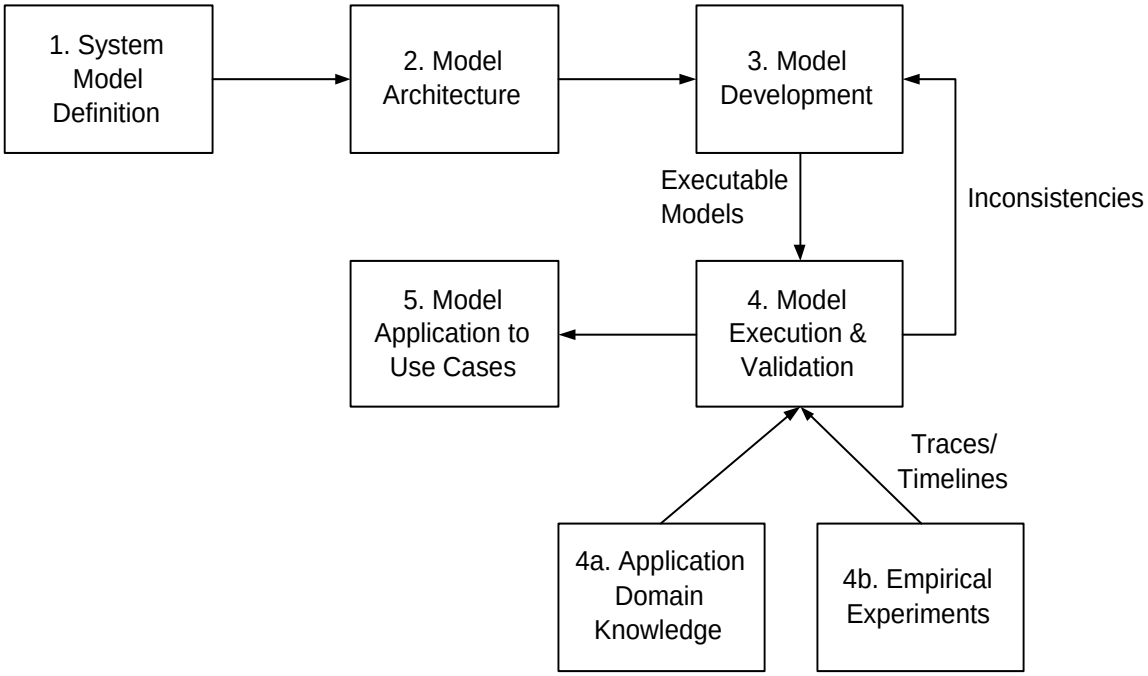


Figure 3.1: Middleware Modeling Methodology

Figure 3.1 illustrates the different steps involved in developing and using our formal models for middleware based applications. Although we discuss its use in the context of ACE, this approach is applicable to a wide range of other sets of reusable system software mechanisms and frameworks. To illustrate how our approach would apply in general, we now describe a methodology we followed in the context of ACE. (1) We first define a system model where we identify the key software mechanisms and their relationships and interactions. (2) We then develop an architecture that forms the foundation for developing an executable model. The architecture is generic enough

that it can be realized and evaluated using different modeling and model checking tools, but fine-grained enough to capture important timing and liveness properties. (3) The next step is to realize this architecture by building concrete models of foundational middleware building blocks using a specific timed automata modeling tool like UPPAAL or IF. The resulting models are executable and amenable to model checking. (4) The executable models then must be validated (*i.e.*, "debugged") to eliminate any inconsistencies. This validation is done by applying these models to representative examples that capture key behavioral semantics arising from different middleware configurations. Two aspects are involved in validating the models - (4a) domain expertise that can be used to derive informal conclusions, and (4b) execution traces from actual runs of these examples using the built middleware. Execution traces can be used to generate detailed timelines and both traces and timelines from the runs can be used for comparison against the traces and timelines obtained by executing the models. Any inconsistencies detected are used in making the necessary modifications to the models and this process is repeated throughout the development/verification process. (5) Finally the middleware models are used to model application use-cases.

In this chapter, we introduce the system model (1) and the modeling architecture (2). In Chapters 4 and 5, we discuss model development (3) in UPPAAL and IF respectively. Model execution and validation (4) are discussed over Chapters 6 and 7. Finally we present two case studies applying (5) the built models in Chapters 8 and 9.

3.1 System Model and Problem Definition

Patterns and pattern languages have been very influential in the design and development of a variety of software systems. Specifically the POSA2 [97] patterns describe in great detail, reusable patterns for building distributed and concurrent middleware. Even though the POSA2 patterns use formal notations like UML interaction diagrams, a more mathematical treatment of the behavioral interaction between the key players in the POSA2 patterns is necessary for formal analysis of middleware that is built using frameworks like ACE [51] that reify these patterns. In this dissertation, we describe the necessary details to support such formalization. The system model that

we describe here is the beginning of such an effort that establishes the scope of this dissertation, although as part of future collaborative work, described in Chapter 10, we intend to enhance this model further in terms of both the formalization and the type of model elements.

Our system model[1] can be expressed as a 6-tuple $\{E, H, I, R, A, \theta\}$, consisting of the following elements:

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

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

[1]The author proposed the initial version of this model, and then enhanced and refined it in collaboration with Dr. Chris Gill (Washington University) and Dr. Henny Sipma (Stanford University)

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

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

Further subdivision of the thread pools may be useful for some kinds of middleware, for example to model thread pools at different priorities within a reactor [81], but for the scenarios considered in this dissertation, we will assume a single thread pool per reactor unless we indicate otherwise. It is also possible for an endsystem to employ additional threads that are not assigned to a reactor, but a useful middleware design idiom is only to use threads from reactor thread pools so that it is easier to apply policies such as prioritization consistently across all threads. Therefore, we confine our attention in this dissertation mostly to the case where all endsystem threads are in reactor thread pools. The only exception to this is the Gateway example described in Chapter 9, where active object [60] threads are used as worker threads as part of the gateway example implementation using the Half-sync Half-async [97] pattern.

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

- $registered : E \times I \times R \times T \rightarrow 2^H$. The set of event handlers registered for event e on interaction channel i in reactor r at time t is given by $registered(e, i, r, t)$.
- $active : R \times T \rightarrow 2^E$. The set of events that have arrived at reactor r but have not been dispatched to event handlers at time t is given by $active(r, t)$.
- $ready : E \times R \times T \rightarrow 2^I$. The set of interaction channels for which event e is active in reactor r at time t is given by $ready(e, r, t)$, and a single event-specific action, such as one read from a socket for a "data ready" event, can be taken on a ready channel without blocking the thread in which that action is taken.
- $dispatched : R \times T \rightarrow 2^\theta$. The set of threads in $threadpool(r)$ that are currently in use to dispatch events to event handlers in reactor r, and thus are not available to dispatch other events from $active(r, t)$ at time t is given by $dispatched(r, t)$.
- $blocked : R \times T \rightarrow 2^\theta$. The set of threads in $threadpool(r)$ that have taken blocking actions that will only unblock and allow the thread to continue when a specific event occurs is given by $blocked(r, t)$. Note that for some scenarios, such as a thread scheduling a timer and then blocking on the timer's expiration, unblocking will not depend on an action being performed in another thread; for other scenarios, such as a thread performing a blocking read on a socket, an event to trigger unblocking must be generated by an action taken by another (possibly remote) thread.
- $deadline : \mathcal{N} \times E \rightarrow T$. The time by which the n^{th} occurrence of event e is constrained to occur is given by $deadline(n, e)$. Event occurrences that do not have timing constraints are assumed to have a deadline of ∞.
- $occurred : \mathcal{N} \times E \rightarrow T$. The time at which the n^{th} occurrence of event e happened is given by $occurred(n, e)$.
- $live : R \times T \rightarrow 2^\theta$. The set of threads assigned to reactor r within each of which at least one action occurs after time t is given by $live(r, t)$.
- $arrival\_time : \mathcal{N} \times E \times R \times I \rightarrow T$. The time of arrival of the nth occurrence of event e on channel i at reactor r is given by $arrival\_time(n, e, r, i)$. Event occurrences are numbered globally rather than by interaction channel. If the nth occurrence of an event happened in a different channel than the one given to the $arrival\_time$ function then the return value would be ∞ indicating that the event never occurred there.
- $dispatch\_time : \mathcal{N} \times E \times R \times I \rightarrow T$. The time of dispatch of the nth occurrence of event e on channel i by reactor r to the appropriate event handler is given

by $dispatch\_time(n, e, r, i)$. If the nth occurrence of an event happened in a different channel than the one given to the *dispatch_time* function then the return value would be ∞ indicating that the event never occurred there.

Problem definition. Our approach hinges on the idea that interference occurs when the actions taken by endsystem threads can affect each other in ways that produce adverse consequences for the system's specified constraints. In Chapter 1, we gave specific examples of interference occurring in ORB middleware. In this research, we address the specific problem of detecting interference in which threads' actions on reactors, event handlers, and interaction channels in the endsystem middleware can cause violations of application-specific timing and liveness constraints.

Forms of Interference. We analyze two forms of interference - blocking delays and exhaustion of threads in a reactor thread pool.

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

Our analysis depends both on (1) the application constraints and (2) the mechanics of the middleware mechanisms that shape the interference. We model the constraints as temporal logic statements and model the middleware mechanisms as timed automata, and use model checking to evaluate whether or not the constraints are always satisfied. Specifically, we use model checking to search for states of the system in which two particular kinds of constraint violations appear: *missed deadlines*, which are timing constraint violations that can occur even when liveness is preserved, and *deadlock* which is a liveness constraint violation that usually also leads to timing constraint violations in subsequent system states. Checking for a missed deadline in a state can be done using our system model by comparing the time at which the

nth occurrence of event e happened, to the deadline for that occurrence of the event: $occurred(n, e) > deadline(n, e)$. Deadlocks can be detected in our system model by determining whether or not we reach a state with global time t after which no further action will be taken by any of a reactor r's assigned threads : $|live(r, t)| = 0$. Note that it is not sufficient to check whether or not all threads in a reactor are blocked: $|blocked(r, t)| = |threadpool(r)|$ says only that no actions can be taken by the threads assigned to reactor r from time t until a subsequent occurrence of an event (*e.g.,* due to an action in a remote thread) causes one of those threads to unblock, and only indicates deadlock if no such event occurs after time t.

When a state containing a constraint violation is reached, the model checker can then produce a trace of the system states that led up to that constraint violation. By examining these traces and correcting the particular patterns of interference they reveal, we can remove design and implementation errors, and also gain insights into new techniques that can in some cases prevent, or in others at least help avoid, those errors. In Chapters 4 and 5 we describe the models we have developed in the research presented in this dissertation, and our use of the modeling and model checking tools within which we represent and explore them. In Chapters 6, 8 and 9 we present case studies showing how different forms of interference can arise, and how model checking can be used to detect them or to verify their absence.

3.2 Middleware Modeling Architecture

Figure 3.2 shows our modeling architecture which forms the basis for our model development (step 3 in Figure 3.1) which is discussed further in Chapters 4 and 5. Our models are divided into three layers: (1) models of network and OS level abstractions such as channels for interprocess communication; (2) models of semantically rich middleware building blocks like threads, event handlers, and reactors; and (3) models of the application functionality implemented in the form of event handlers. Although Figure 3.2 shows a static view of our models, the realizations of these models using IF or UPPAAL are executable and can be model-checked against system property specifications. The unshaded rectangular boxes shown in Figure 3.2 are modeled using

timed finite state automata specified using a modeling language that supports timed automata, *e.g.,* IF or UPPAAL.

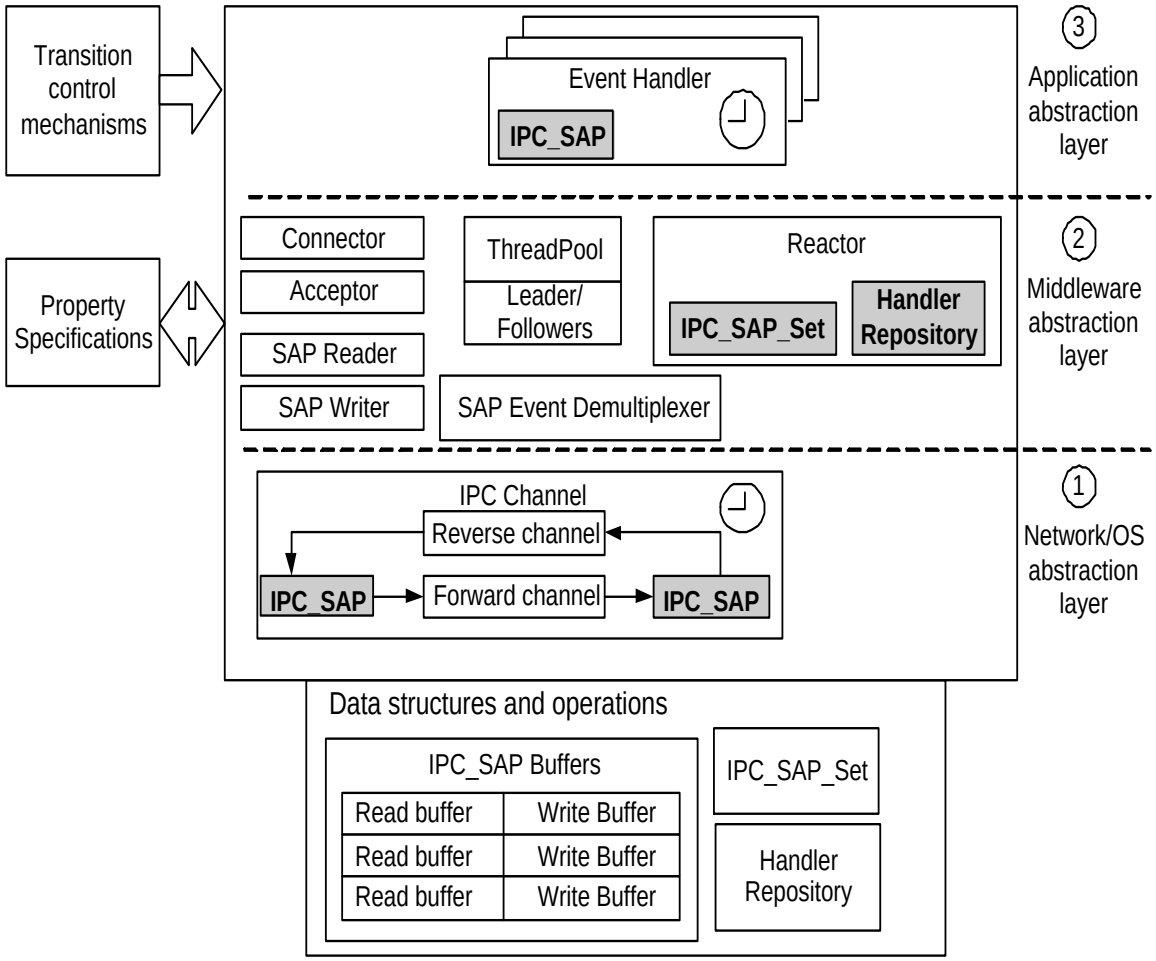


Figure 3.2: Middleware Modeling Architecture

To reduce the complexity of the state space that must be explored by model checking, we also abstract certain details of the system out of the timed automata models themselves. The shaded rectangular boxes shown in Figure 3.2 are foundational data structures and associated operations that we have implemented to complement the timed models. Some of these foundational data structures are shared across automata, and thus form a basis for event-driven communication between automata. For example, one automaton could wait for data to be updated in a shared data structure (*e.g.,* IPC_SAP_Buffers) and another automaton could update this shared

data structure thus allowing the waiting automaton to proceed further. Communication between automata occurs through specific interfaces to the shared data. Using these interfaces, the shared data can be manipulated. Timed transitions (transitions that are guarded with conditions based on clock variables) are indicated in Figure 3.2 by timer icons. We now describe each of the layers of our middleware modeling architecture in greater detail.

Network/OS abstraction layer. At the lowest architectural layer, we model inter-process communication (IPC) mechanisms - such as sockets, pipes, FIFOs, and message queues - as IPC channels. An IPC channel has two Service Access Points (SAPs), for convenience called the left-hand-side SAP (lhs-SAP) and the right-hand-side SAP (rhs-SAP). Each SAP has a read-buffer and a write-buffer associated with it. The read-buffer is used by the SAP to receive any data sent to it from another SAP and the write buffer is used to send data from that SAP to another SAP. Based on the features offered by a model checker (*e.g.,* Abstract Data Types in IF), one may choose not to expose these buffers to the model checker if the data itself does not play a significant role in the kind of properties, *i.e.*, timeliness and liveness requirements, with which our research is concerned. Instead, the read and write buffers associated with each SAP may be stored in IPC SAP buffers outside of the model. Each SAP has a unique handle associated with it and this handle is used in the IPC_SAP_Buffers data structure to access the data buffers associated with a SAP. For example, the IF model checker provides the Abstract Data Type feature which we use to implement the data structures and operations and the states of these data structures are not exposed to the model checker. We describe this in detail in Section 5.2.

An IPC channel is bidirectional. It is modeled, however, as two data-transfer automata, one for the forward direction, and one for the reverse direction. The forward channel automaton waits for data to be enqueued on the write-buffer of the lhs-SAP and transfers it to the read-buffer of the rhs-SAP. The reverse channel automaton waits for data to be enqueued on the write buffer of the rhs-SAP and transfers it to the read-buffer of the lhs-SAP. These forward and reverse channel automata also can be parameterized with propagation delays, if needed. These propagation delays are implemented using transitions guarded with clock variables in the model. Modeling the IPC channel as an automaton with those parameters allows us to model different

degrees of asynchrony and non-determinism introduced by real-world communication channels.

Middleware abstraction layer. The next architectural layer above the network/OS layer is the middleware layer, where we model abstractions of semantically rich middleware building blocks. Here we use the foundational data structures and operations to encapsulate data structures like the event handler repository used by the reactor to store mappings between a Service Access Point (SAP) and the handler associated with that SAP. This table is populated whenever an event handler is registered with a reactor.

Each middleware primitive is modeled so that the behavior seen when the model is executed closely adheres to that of the actual implementation. This faithful modeling of the middleware primitives in turn results in high-fidelity models of higher-level middleware services, obtained by composing these primitive models.

Application abstraction layer. In our models, we encapsulate the application functionality using event handlers as is customary when developing ACE applications in practice [95, 96]. Each event handler reads data from or writes data to IPC channels, which in turn model interactions between different event handlers. The computation performed by an event handler is abstracted away and represented by a single transition, guarded by a constraint on a clock variable to model its execution time as necessary. An event handler reads and writes data to an SAP using the foundational operations described before.

Property specifications for verification. Once we build models of applications using the layered approach that we have just described, the timing and liveness properties should be specified using a suitable formalism. This formalism depends on the model checking tool within which our models are executed. For example, in this dissertation, we use formalisms based on IF (observers) and UPPAAL (CTL expressions) to express timing and liveness properties.

Transition Control Mechanisms. Since model checking tools search the state space exhaustively for any violation/satisfaction of system properties, for a large state space, the technique may become intractable. To make model checking more tractable, the search space should be restricted by pruning out unnecessary non-determinism. This is possible by providing middleware domain specific state space optimizations.

3.3 Summary

In this chapter, we have described a middleware-level system model that captures the key middleware elements that form the focus of this dissertation, and their relationships. We proposed a middleware modeling architecture that forms the basis for the models developed in this dissertation.

Chapter 4

Models in UPPAAL

In Chapter 3, we described the different steps involved with building and using the middleware models that have been developed in this dissertation. In this chapter, we focus on model development (step 3) using UPPAAL [7], a tool that supports timed automata modeling and model checking. We reify the modeling architecture described in Section 3.2 using the modeling constructs in UPPAAL.

4.1 Realization of the Middleware Modeling Architecture in UPPAAL

Figure 4.1 shows our realization of the modeling architecture discussed in Section 3.2, in UPPAAL. Note that the generic architecture shown in Figure 3.2 has been made more concrete in Figure 4.1 using modeling constructs in UPPAAL. For example, transition control mechanisms are realized in UPPAAL using techniques for maximal progress which we discuss in Section 4.3.1. Property specification is done through CTL [19] expressions. Formal models of foundational middleware building blocks are realized using timed automata in UPPAAL. We developed foundational data structures like IPC_SAP, collections of IPC_SAPs, handler repositories, *etc.* using array and user-definable record data structure features in UPPAAL. We developed functions to operate on these data structures so that these functions can be accessed from the timed automata models in UPPAAL. The ability to define and call user-defined functions was introduced in UPPAAL 3.6 Alpha 1(Nov 2005). Note that these user-defined functions facilitate carrying out data structure operations like adding a handler to

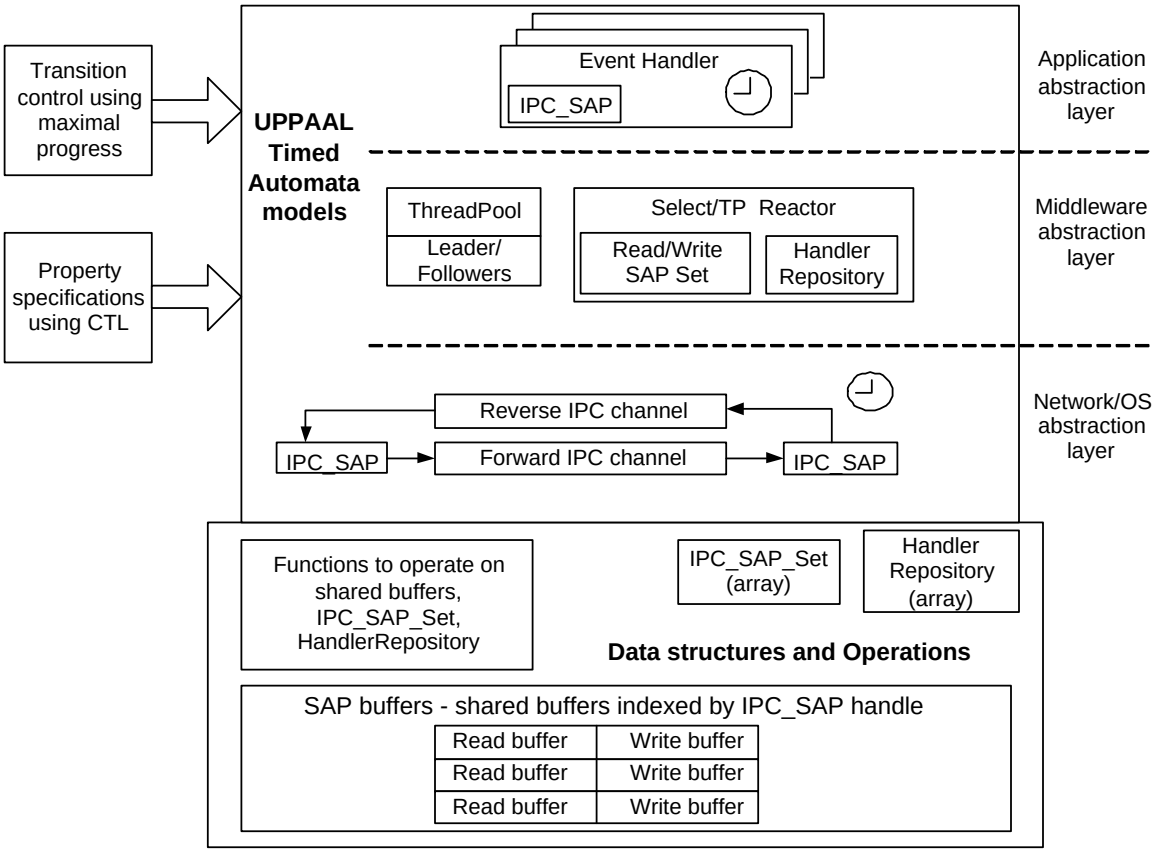


Figure 4.1: Realization of the Modeling Architecture in UPPAAL

the handler repository. These user-defined functions are essentially a vehicle to carry out operations on a subset of the system state, which if done using automata (as opposed to functions) could result in an unnecessary increase in the number of states and transitions. UPPAAL 3.6 also offers simple control structures like for, while, if and return that can be used within user-defined functions. User-defined functions in UPPAAL cannot have state transition constructs within them. They can be used only for data transformation activities and hence can only be called as part of the action statements in an exectuable automata model. It is worth noting that the language in which the a function is defined in UPPAAL is very similar to C, although it is not C. In contrast, the IF modeling tool that we will use in Chapter 5 offers a procedure construct that wraps actual C code. Note that some of the middleware building blocks (Acceptor, Connector) from the architecture shown in Figure 3.2 are missing from Figure 4.1. These are not implemented because of the inability in UPPAAL to create automaton dynamically, which is necessary to model these building blocks. For example, an Acceptor creates a service handler dynamically when a connection has been established.

4.2 Modeling Foundational Data Structures and Functions

We now explain the foundational data structures and functions that we developed and which are used by models of middleware building blocks and the applications based on them. This discussion establishes a basis for our subsequent discussions about the middleware building block models which use these foundational functions and data structures. Figure 4.2 shows some of the foundational data structures modeled using UPPAAL constructs.

The IPC_SAP_Buffer structure models the read and write buffers associated with a service access point (SAP). The collection of all SAPs in the system is modeled by the array ipc_sap_buffers. Each IPC_SAP structure has a unique handle associated with it, which is used to index the ipc_sap_buffers array to get to the buffers associated with that IPC_SAP. Note that we maintain only the number of bytes enqueued in a buffer

```
typedef struct
{
  int readq;
  int writeq;
} IPC_SAP_Buffer;

IPC_SAP_Buffer
   ipc_sap_buffers[MAX_IPC_SAP_BUFFERS];

typedef struct
{
  int handle;
  bool suspended;
  int annotation;
} IPC_SAP;

typedef struct
{
  IPC_SAP members[MAX_MEMBERS];
  int currsize;
} IPC_SAP_Set;
```

```
typedef struct
{
  IPC_SAP sap;
  int eh;
} HandlerRepoItem;

typedef struct
{
  HandlerRepoItem
      members[MAX_REPO_HANDLERS];
  int currsize;
} HandlerRepo;

typedef struct
{
  bool reactor_in_use;
  HandlerRepo handler_repo;
  IPC_SAP_Set read_sap_set_to_watch;
  IPC_SAP_Set write_sap_set_to_watch;
  int handle_events_stack_depth;
} Reactor_State;
```

Figure 4.2: A Sampling of Foundational Data Structures

and not the actual data, which in turn reduces the state space that must be checked without modifying the outcome of verification since we limit our attention to the effects of concurrency semantics in middleware on the timing and liveness properties of the application. The ipc_sap_buffers array is a global variable in our UPPAAL model, which is shared across all automata. Communication between automata is achieved by putting bytes into the shared buffers, which in turn will trigger other automata that are waiting for bytes to be enqueued into specific buffers. The HandlerRepo data structure is used to keep track of the association between service access points and event handlers. Each reactor automaton uses the Reactor_State data structure to maintain its state so that automata modeling threads in a reactor thread pool can all access the state of the reactor.

Figure 4.3 shows a sampling of the UPPAAL functions that we have developed to manipulate the global shared data structures. Functions named starting with prefix ipc_set_ operate on an IPC_SAP_Set. For example, ipc_set_suspend_sap is used to mark a SAP as suspended by setting the suspended flag in an IPC_SAP object. This flag is used by a reactor to determine whether an SAP should be included in its set of SAPs to watch for I/O events. ipc_set_resume_sap resets the suspended flag. The

```
//Operations on IPC_SAP_Set
int ipc_set_add_member(IPC_SAP_Set &sap_set, IPC_SAP sap)
int ipc_set_size(IPC_SAP_Set ipc_set)
int ipc_set_pop_first(IPC_SAP_Set &sap_set, IPC_SAP& first_sap)
int ipc_set_suspend_sap(IPC_SAP_Set &sap_set, IPC_SAP& sap)
int ipc_set_resume_sap(IPC_SAP_Set &sap_set, IPC_SAP& sap)
IPC_SAP_Set ipc_set_get_non_suspended_saps(IPC_SAP_Set &sap_set)
int ipc_set_pop_first_non_suspended(IPC_SAP_Set &sap_set,
                                    IPC_SAP& first_non_susp_sap)
int ipc_set_clear(IPC_SAP_Set &sap_set)
IPC_SAP_Set get_hot_read_saps(IPC_SAP_Set &read_sap_set)
IPC_SAP_Set get_hot_write_saps(IPC_SAP_Set &write_sap_set)

//Operations on queues associated with IPC SAPs
int init_ipc_queues()
int put_data(IPC_SAP sap, int qtype, int bytes)
int get_data(IPC_SAP sap, int qtype)
int queue_level(IPC_SAP sap, int qtype)
bool is_empty(IPC_SAP sap, int qtype)
bool is_any_sap_hot(IPC_SAP_Set read_sap_set,
                    IPC_SAP_Set write_sap_set)

//Operations on handler repository
EH_PID HR_get_handler(HandlerRepo &handler_repo, IPC_SAP sap)
int HR_add_handler(HandlerRepo &handler_repo, IPC_SAP sap, EH_PID eh)
```

Figure 4.3: A Sampling of Foundational Functions

ipc_set_get_non_suspended_saps function is used to filter a set of SAPs and extract only the ones for which the suspended flag is not set. Functions put_data and get_data are used to enqueue and dequeue bytes into and from the appropriate read and write buffers in the ipc_queues global shared data structure. These functions use the passed IPC_SAP object's handle to index the ipc_queues and increment or decrement the byte counters (the readq and writeq variables associated with the specified IPC_SAP object. The is_any_sap_hot function models the *select* and *WaitForMultipleObjects* OS calls. It takes it a set of SAPs and determines whether any of these are "hot" with an I/O event. For an SAP in the read set, "hot" means the read buffer for that SAP is not empty. For an SAP in the write set, "hot" means the write buffer for that SAP is not full. The functions get_hot_read_saps and get_hot_write_saps gets the SAPs that are read-ready and write-ready from a set of SAPs. These are typically used by a reactor to get the "hot" SAPs from the set of SAPs that are watched by the reactor. Finally, functions named starting with prefix HR_ are used to manipulate a reactor's handler repository.

4.3 Modeling Issues in UPPAAL

In this section, we identify several engineering challenges we faced in building our models in UPPAAL and then present solutions to overcome these challenges.

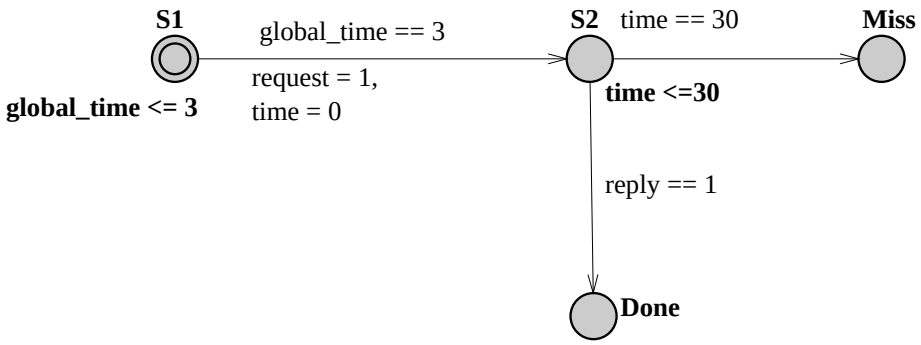
4.3.1 Maximal Progress in UPPAAL

Before we discuss our implementation of middleware building block models in UPPAAL, we first consider the idea of maximal progress in UPPAAL, its consequences, and how we work around particular problems posed by it. The solutions to these problems are used throughout our models. First, in UPPAAL, communication between two automata can be achieved through *rendezvous* synchronization, broadcast synchronization and/or shared variables. In *rendezvous* synchronization, there is a handshake between two automaton on the same channel, with one automaton waiting on a channel (denoted as channel?) and another automaton sending on the same channel (denoted as channel!). The sending action won't be enabled unless there is a

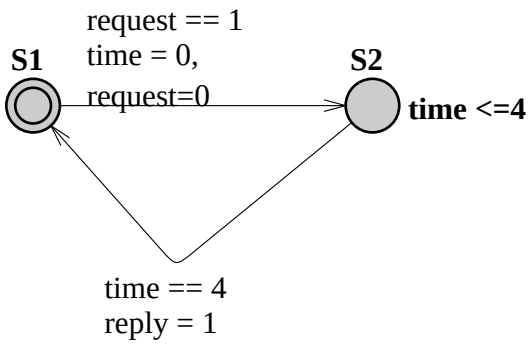
receiver waiting on the same channel on which the sender is sending, and vice-versa. A channel in UPPAAL, by default, is a *rendezvous* channel. A channel also can be qualified as a *broadcast* channel, in which case multiple receivers can listen to and receive messages sent by a single sender on the same channel. In this case, unlike with a *rendezvous* channel, the sender can send on a channel even if there are no receivers listening on that channel. The channels do not maintain history and hence a message sent on a broadcast channel will be lost unless a receiver is already listening on that channel.

Many event-triggered systems need to communicate asynchronously by means of events that are triggered by change of some state variable. UPPAAL supports the notion of guarded transitions, where an automaton can wait on some state-variable-based condition to be satisfied. Guarded transitions can be used to model state transitions in event-triggered systems based on state variable changes. We now explain the problems arising from the lack of maximal progress semantics in UPPAAL, in the presence of both time and event based transitions.

Figures 4.4(a) and 4.4(b) show the client and server automata of a simple example to illustrate the need for maximal progress in time and event based modeling of systems. In this example, we are modeling a very simple client/server example, where the client sends a request to a server at global_time = 3 and the server sends a reply back to the client after doing some computation for 4 time units after it detects the request. Note that there are three clock variables that are being used in this example - global_time, Client.time and Server.time. global_time is a global variable that is accessible to both the client and server automata. Client.time and Server.time are local clock variables in the client and server automata respectively. These local clock variables are typically used to keep track of elapsed or relative time. For example, a local variable time is used in the Client automaton to model the fact that if the client does not get the reply back within 30 time units (a relative deadline of 30 time units from the time when the request was sent), there is a deadline miss and a state transition is made to state Miss. Local clock variables can be reset as seen in Figure 4.4(a), where during the state transition Client:S1→S2, the local clock variable time is reset to 0. All clock variables in UPPAAL are synchronized and are advanced by UPPAAL atomically.



(a) Client



(b) Server

Figure 4.4: Maximal Progress Example

If, on the other hand, a reply is received before the deadline, the client takes a transition to the Done state. This example is typical of time and event based systems, where triggering of state transitions could be based on both time (*e.g.,* request to be sent at global_time = 3) and event (*e.g.,* request arrival at the server denoted by request == 1).

Note that there is an invariant in state S1 of the client automaton (global_time $\leq$ 3). This forces a state transition out of that state at global_time = 3. Not having this invariant could allow UPPAAL to advance time without leaving that state, which is not the behavior that we want. A similar purpose is served by the invariants in state S2 of the client automaton and state S2 of the server automaton. Shared variables such as request and reply are used for indirect communication between the client and server automata. The client updates the shared variable request to 1 and the server waits for this variable to become 1. Similarly, the server updates the shared variable reply to 1 and the client waits for this variable to become 1 as it waits for its reply.

When we run a simulation of this composed system in UPPAAL, we notice that the first transition in the system is Client:S1→S2. At state S2, there are two transitions enabled in the UPPAAL model checker - Client:S2→Miss and Server:S1→S2. The transition Server:S1→S2 is enabled because the value of the request variable is 1. However the transition Client:S2→Miss is also enabled because of lack of *maximal progress* or *as soon as possible* semantics in UPPAAL. With maximal progress semantics, a model checker tries to make as much progress as possible, by taking successive *non-time-based* transitions, until no more *non-time-based* transitions are enabled and only then advances time. Since maximal progress semantics is not implemented in UPPAAL, advancing time is also a possibility at any point in UPPAAL and hence the model checker can give a false positive for a deadline miss query like E◇ Client.Miss, which asks whether there is any path in the state space where in any state the Client automaton is in the Miss state.

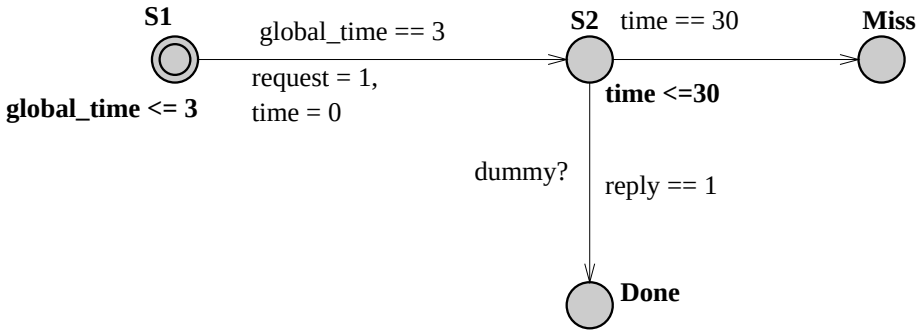
4.3.2 Constraining the State Space with Maximal Progress

Not only does the lack of maximal progress semantics cause the UPPAAL verifier to give false positives as was discussed in the previous section, but it also makes

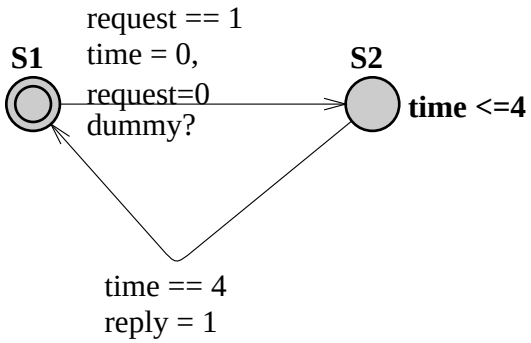
the state space larger than necessary by introducing inappropriate non-determinism, where time based transitions are also enabled apart from the shared variable based guarded transitions.

We now describe how we constrain the state space by forcing UPPAAL to achieve maximal progress and hence eliminate this non-determinism. In the example in Section 4.3.1, one intuitive solution is to mark the state Server:S1 as an *urgent* state (in UPPAAL, time cannot progress in an *urgent* state). In other words, the model checker has to execute enabled transitions leaving from an *urgent* state before it can advance time. Unfortunately this solution won't work because of the very same issue that it is supposed to address - progress of time. If we mark the state Server:S1 as *urgent*, then the model checker will try to execute the sole transition Server:S1→S2 before executing any other transition since S1 is one among the initial states of the composed automata. The transition Server:S1→S2 would become enabled only when request = 1, which will happen only when time progresses to 3 and the Client automaton updates request to 1. However time cannot progress in an urgent state and hence no transitions are enabled, resulting in a deadlock.

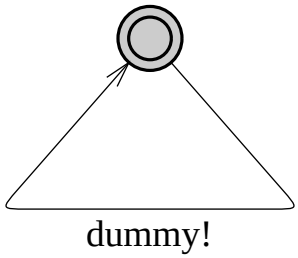
Our solution to these problems is based on the concept of an *urgent channel* in UPPAAL used in conjunction with untimed guards [7] and *rendezvous* synchronization. This solution applied to the example in Section 4.3.1 is shown in Figure 4.5. Every shared-variable based guarded transition carries a *rendezvous* synchronization with a dummy channel (denoted as dummy?). A new automaton is introduced in the system, which has a single state and a single transition that has a corresponding channel send (denoted dummy!). The dummy channel is marked as urgent so that time does not progress while executing this *rendezvous*. In our modified example in shown in Figures 4.5(a)- 4.5(c), the dummy *rendezvous* is introduced in the two transitions whose guards are based on shared variable updates - `Server:S1→S2` and `Client:S2→Done`. As soon as request becomes 1, `Server:S1→S2` becomes enabled and since time cannot progress in this transition because of the rendezvous synchronization on an urgent channel, this is the only transition that is enabled. We use this solution in our models in the context of all shared variable based guarded transitions. One potential limitation [7] to the use of urgent channels is that they cannot be used in conjunction with timed guards, which in our case is not a problem since we are using them only in conjunction with untimed guards.



(a) Client



(b) Server



(c) Dummy

Figure 4.5: Maximal Progress Solution

4.4 UPPAAL Models of Middleware Building Blocks

We now describe the models of middleware building blocks that we developed in UPPAAL. These models use the foundational data structures and operations that we described in Section 4.2. They also use the solution for maximal progress described in Section 4.3.2 so that communication among automata through shared variables take precedence over progress of time.

4.4.1 IPC Channel

We model an IPC Channel as a combination of two automata - a forward channel and a reverse channel. Each of these channels is modeled as shown in Figure 4.6. The SAPs and channels are typically instantiated in UPPAAL as follows:

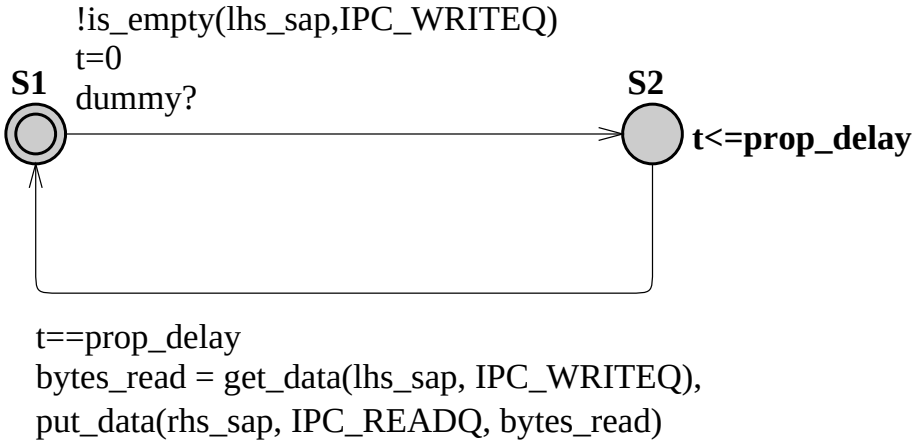


Figure 4.6: Model of an IPC Channel

In effect, this automata transfers bytes from one slot (handle in lhs-SAP) in the ipc_sap_buffers data structure to another slot (handle in rhs-SAP). This automata waits for data to appear in the lhs-SAP buffer. The wait is modeled by the is_empty function which takes the lhs-SAP object as an argument. At state S1, the automaton waits for data on the write buffer of the lhs-SAP. When there is some data in the buffer, the automaton changes state to S2. Note the usage of dummy? once again

to achieve maximal progress. At S2, the automaton waits for a specified number (prop_delay) of time units. The value for prop_delay is passed as the third parameter as is shown in the UPPAAL model instantiation in Figure 4.7. For example,

```
//process template arguments for IPC_Channel automaton
IPC_Channel(IPC_SAP &lhs_sap, IPC_SAP &rhs_sap, int prop_delay)

//in the global declarations section in UPPAAL modeler
IPC_SAP ipc_sap_0 = {0, false, 0};
IPC_SAP ipc_sap_1 = {1, false, 0};

//in the Process assignment section in UPPAAL modeler
client1_eh1_fwd = IPC_Channel(ipc_sap_0,ipc_sap_1,5);
client1_eh1_rev = IPC_Channel(ipc_sap_1,ipc_sap_0,7);
```

Figure 4.7: Instantiation of IPC Channel Automaton

channel client1_eh1_fwd transfers data from the write buffer of ipc_sap_0 to the read buffer of ipc_sap_1, with a propagation delay of 5. The corresponding reverse channel client1_eh1_rev transfers data from the write buffer of ipc_sap_1 to the read buffer of ipc_sap_0. After waiting for the specified delay period, the automaton enqueues the data to the read buffer of the rhs-SAP and comes back to state S1, and listens again for data to be enqueued.

4.4.2 Select Reactor

Figure 4.8 shows a model in UPPAAL of the ACE Select Reactor. Note that in this model we extensively use the foundational functions that we discussed in Section 4.2. The reactor is modeled as a process template with parameters and instantiation as is shown in Figure 4.9.

The reactor is modeled as a passive object waiting for the handle_events method call from the reactor thread (see Figure 4.10). The handle_events method call is modeled as a rendezvous channel in UPPAAL. Similarly, the method call's return is modeled as another UPPAAL channel handle_events_return. These channels are passed as process template parameters, as shown in Figure 4.9 to the Select_Reactor automaton in UPPAAL so that multiple instances of the Select_Reactor automaton can be created with different channels for communication with other automata in the

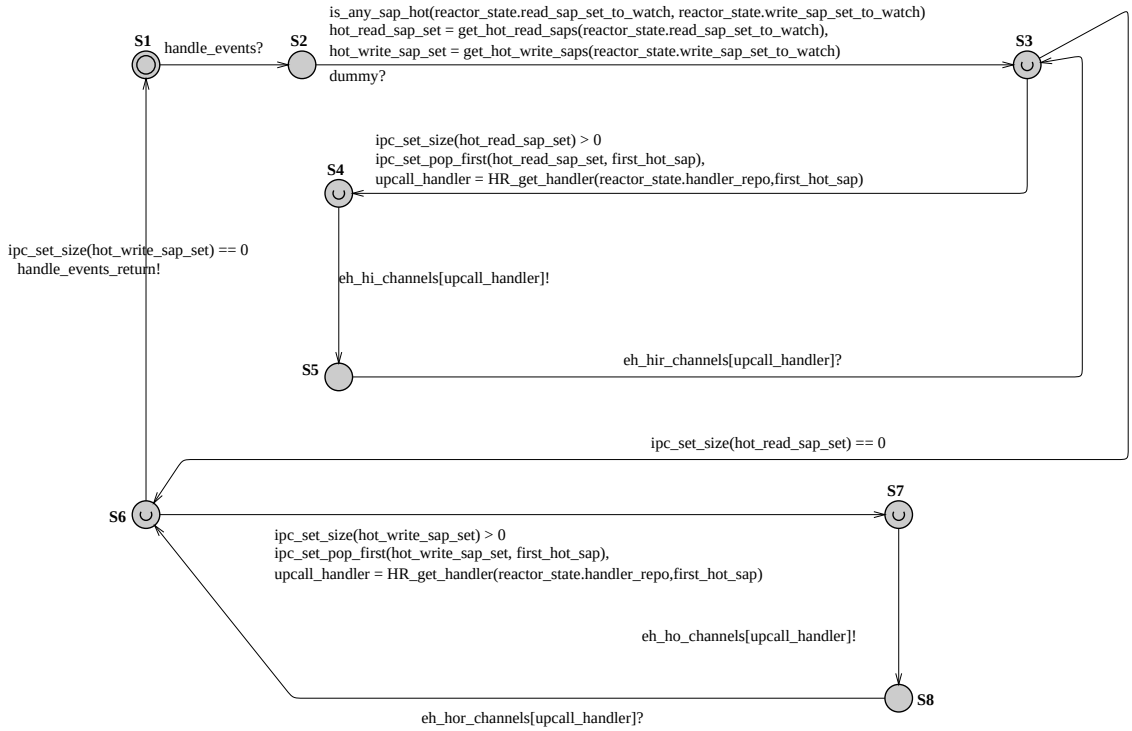


Figure 4.8: Model of Select Reactor

```
//process template arguments for Select_Reactor
Select_Reactor(THREAD_ID thread_id, urgent chan &handle_events,
urgent chan &handle_events_return, Reactor_State &reactor_state)

//instantiation of Select_Reactor automaton
reactor1 = Select_Reactor(REACTOR1,
                          handle_events_channels[REACTOR1],
                          handle_events_return_channels[REACTOR1],
                          reactor_states[REACTOR1]);
```

Figure 4.9: Instantiation of Select Reactor Automaton

composed system. REACTOR1 is an integer constant used to identify each reactor in the system. Each reactor has a reactor state associated with it, *e.g.,* handler repository, SAP handle set to listen on. These data structures are maintained in the global state space and are passed as template arguments to each reactor instantiation as is shown in Figure 4.9.

We now discuss in detail the state transitions in the Select_Reactor model illustrated in Figure 4.8. At state S1, the reactor waits for the handle_events method call from the reactor thread. On receipt of the handle_events event, the reactor moves to state S2. In state S2, the reactor checks to see whether any of the SAPs that it is watching are ready ("hot") for reading or writing. The reactor waits in S2 until there is at least one ready SAP. It then gets the set of SAPs that are read-ready and the set of SAPs that are write-ready. In state S3, the reactor checks whether there are read-ready SAPs. If so, it removes the first SAP from the read-ready set of SAPs and obtains the event handler channel corresponding to that SAP from the handler repository and then moves to state S4. It uses the event handler channel to *rendezvous* with that event handler (S4→S5). Note that the channels to communicate with the event handlers are stored in the global arrays - eh_hi_channels for handle_input method calls, eh_hir_channels for handle_input method call returns, eh_ho_channels for handle_output method calls and eh_hor_channels for handle_output method call returns . In state S5, the reactor waits for the handle_input method call return from the event handler. This sequence of steps (S3→S4→S5) is repeated until there are no more read-ready SAPs. The same sequence is then repeated for the write-ready SAPs (S6→S7→S8). In this sequence, instead of the handle_input and handle_input_return channels, handle_output and handle_output_return channels are used to communicate with event handler automata.

Figure 4.10 shows the UPPAAL model of a thread that repeatedly calls the handle_events method on the reactor. This models the reactor event loop in an application. Typically, the loop is terminated by an application-specific condition which for simplicity and generality we have not shown here.

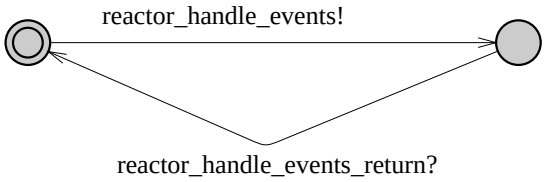


Figure 4.10: Model of a Reactor Thread

4.4.3 Reentrant Select Reactor

In some cases an event handler could call handle_events on a reactor in the context of an upcall, which constitutes a recursive call into the reactor handle_events method. For instance, when using ACE to implement an object request broker like TAO [50] or nORB [104], this is how an event handler waits for a pending reply using the reactor rather than directly waiting on the connection. An example of this was shown in Section 1.2.2. We now discuss in detail how recursive calls can be modeled in UPPAAL, using the Reentrant Select Reactor model shown in Figure 4.11 as an example.

Lack of recursive capability in Select Reactor model. In Figure 4.8, we saw that when the select reactor is in the middle of an upcall to an event handler, it is either in state S5 or S8 depending on whether the upcall corresponds to a read-ready or write-ready SAP. Therefore if a recursive call is made to the handle_events method of the reactor by an event handler as part of the upcall, there will be a deadlock while executing the model since the reactor automaton is not in state S1 where the reactor automaton waits for the handle_events method call. Instead the reactor automaton waits for the event handler to complete the upcall processing and the event handler won't be able to complete its processing until the handle_events call to the reactor completes.

To support such recursive calls, we revisit the notion that the reactor model in Figure 4.8 models the execution call stack of the handle_events method implementation. Each execution stack frame of the handle_events method call can be modeled by an instance of the Select_Reactor automaton. With additional modifications, recursive

calls can be supported. These modifications ensure that these different instantiations of the reactor automaton use different channels for communication with other automata.

Figure 4.11 shows the modifications necessary to the Select_Reactor automaton to support recursive calls. We introduce a new variable as part of the reactor state - handle_events_stack_depth - to keep track of the depth of recursion. As part of the transition S1→S2, we increment this variable by 1 and as part of the transition S6→S1, we decrement this variable by 1. This depth counter is used by an event handler to choose the appropriate channel to communicate with the reactor automaton at the current level of nesting. To facilitate this choice, we need to instantiate multiple instances of the Reentrant_Select_Reactor automaton as shown in Figure 4.12.

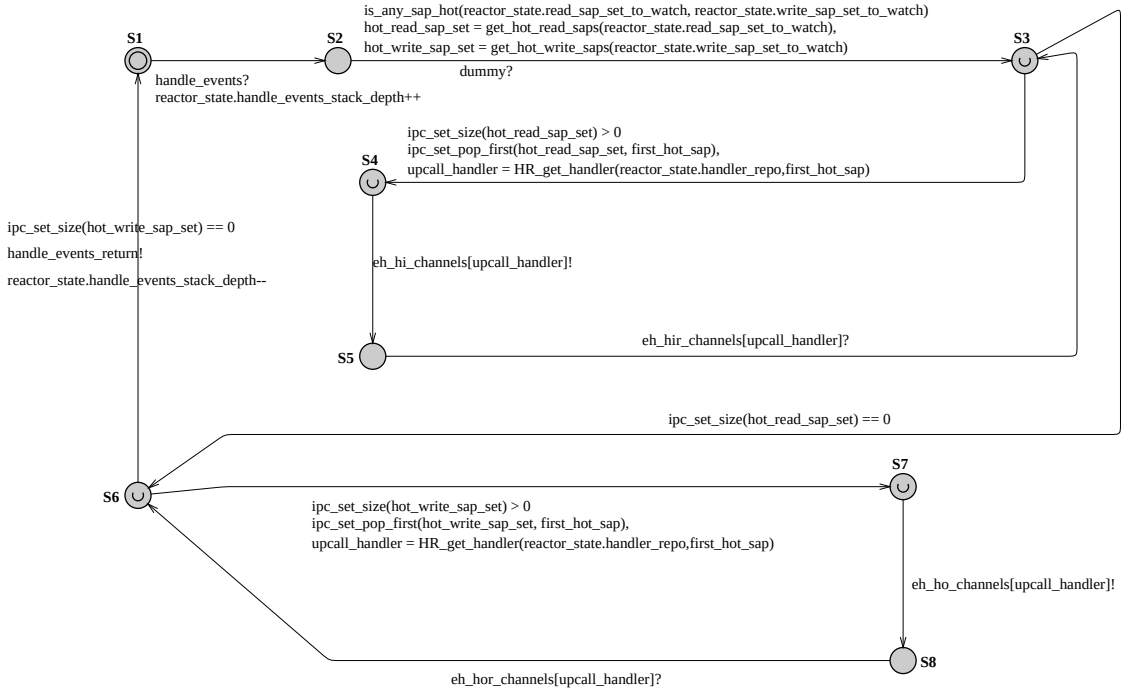


Figure 4.11: Model of Reentrant Select Reactor

Both reentrant_handle_events_channels and reentrant_handle_events_return_channels are global arrays that contain channels for communication, with each instantiated Reentrant_Select_Reactor automaton. Note that the same reactor state is passed to all the instantiations. This is akin to the *this* object reference being passed implicitly as the

```
//execution stack at depth 0
reactor1_s1 = Reentrant_Select_Reactor(REACTOR1,
                reentrant_handle_events_channels[REACTOR1][STACK0],
                reentrant_handle_events_return_channels[REACTOR1][STACK0],
                reactor_states[REACTOR1]);

//execution stack at depth 1
reactor1_s2 = Reentrant_Select_Reactor(REACTOR1,
                reentrant_handle_events_channels[REACTOR1][STACK1],
                reentrant_handle_events_return_channels[REACTOR1][STACK1],
                reactor_states[REACTOR1]);

//execution stack at depth 2
reactor1_s3 = Reentrant_Select_Reactor(REACTOR1,
                reentrant_handle_events_channels[REACTOR1][STACK2],
                reentrant_handle_events_return_channels[REACTOR1][STACK2],
                reactor_states[REACTOR1]);
```

Figure 4.12: Instantiation of Reentrant Select Reactor Automaton

first parameter to an object method implementation in an object oriented language's run-time execution model.

To communicate with the appropriate reactor automaton, the current stack depth is used as shown in Figure 4.13. The current handle_events_stack_depth is obtained from the appropriate reactor's state and then the reentrant_handle_events_return_channels is used to obtain the appropriate channel over which to communicate with that reactor automaton.

```
//obtain the current stack depth
handle_events_stack_depth =
   reactor_states[reactor_pid].handle_events_stack_depth
//wait for return from a handle\_events call on a reactor
reentrant_handle_events_return_channels[reactor_pid][handle_events_stack_depth]?
```

Figure 4.13: Selecting from a Reentrant Select Reactor Stack

4.4.4 ThreadPool Reactor

The ThreadPool Reactor model shown in Figure 4.14, is an extension of the Select Reactor model in which we model multiple threads taking turns to wait in the same

reactor using the Leader/Followers pattern. Multiple instantiations of the TP_Reactor automaton are instantiated as shown in Figure 4.15. This is to simulate the different execution stacks associated with the execution of the reactor handle_events method under different thread contexts.

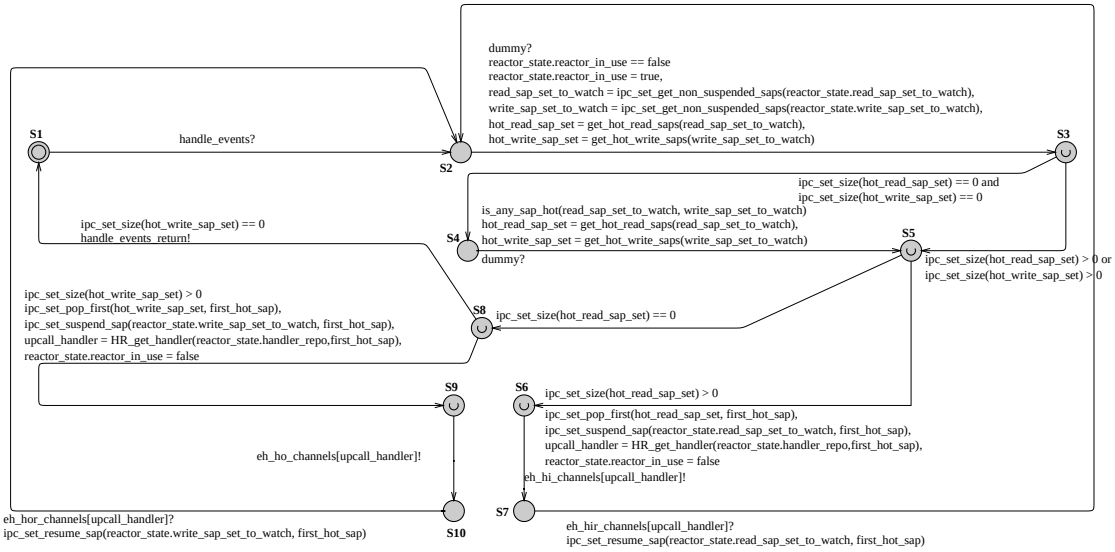


Figure 4.14: Model of ThreadPool Reactor

This use of multiple automata to simulate the execution stacks is very similar to the Reentrant_Select_Reactor model. The only difference is the way in which the channels to the different automata are obtained. In the case of the Reentrant_Select_Reactor model, the current stack depth is used to access the appropriate channel, since the execution is *recursive* in the context of the *same* thread. In contrast, with the TP_Reactor model, execution of handle_events is in the context of *different* threads and hence the thread id is used as an index to access the appropriate channels as shown above. For example, the automaton instance reactor1_t1 uses the channel stored at location handle_events_channels[THREAD1] for its communication and the thread pool thread tp_thread1 is instantiated with the same channel. This approach provides isolation between the call stacks of different threads.

Initially, each thread in a thread pool for a reactor communicates with its own reactor automaton instance using the appropriate handle_events channel. The reactor automaton makes a transition from state S1 to state S2. At S2, we have to model synchronization between the threads in a thread pool that is used to control access to

```
//Execution stacks for two threads in thread pool for REACTOR1
reactor1_t1 = TP_Reactor(THREAD1,handle_events_channels[THREAD1],
                  handle_events_return_channels[THREAD1],
                  reactor_states[REACTOR1]);
reactor1_t2 = TP_Reactor(THREAD2,handle_events_channels[THREAD2],
                  handle_events_return_channels[THREAD2],
                  reactor_states[REACTOR1]);

//Execution stacks for a single thread in thread pool for REACTOR2
reactor2_t3 = TP_Reactor(THREAD3,handle_events_channels[THREAD3],
                  handle_events_return_channels[THREAD3],
                  reactor_states[REACTOR2]);

//threads in thread pool for REACTOR1
tp_thread1 = ReactorThread(handle_events_channels[THREAD1],
                           handle_events_return_channels[THREAD1]);
tp_thread2 = ReactorThread(handle_events_channels[THREAD2],
                           handle_events_return_channels[THREAD2]);

//threads in thread pool for REACTOR2
tp_thread3 = ReactorThread(handle_events_channels[THREAD3],
                           handle_events_return_channels[THREAD3]);
```

Figure 4.15: Instantiation of ThreadPool Reactor Automaton

the reactor event demultiplexing mechanism. We use the reactor_in_use flag to achieve this synchronization.

For example, consider a scenario where multiple threads in a thread pool are at state S2 trying to gain access to the reactor event demultiplexing mechanism modeled by the function is_any_sap_hot. The model checker selects one reactor automaton non-deterministically and allows it to execute the transition S2→S3. This automaton now is designated as the leader automaton that models the leader thread, and all the other automata which use the same reactor are then designated as being follower automata. The leader automaton sets the reactor_in_use flag to true in the reactor's state. When the leader automaton is in state S3, the transition S2→S3 is disabled for all follower automata, since the condition reactor_in_use==true is not true anymore.

After state S3, the working of the leader automaton very closely resembles that of the Select_Reactor automaton with some key differences in the transitions S5→S6 and S8→S9. In both cases, the reactor suspends the SAP using the function ipc_set_suspend_sap before making an upcall to the event handler. Moreover, it resets the reactor_in_use

flag to false so that a follower thread can now gain access to the reactor. Thus we have seen how the reactor_in_use flag can be used as both a critical section lock as well as a condition variable on which the follower threads block waiting to gain control of the reactor's state. One of the follower threads now becomes the leader. Note that the previous leader thread is in the process of an upcall to an event handler and the corresponding SAP has been suspended to avoid simultaneous upcalls to the same event handler from a different thread. The newly elected leader does not include any suspended SAPs in its list of SAPs to watch. This filtering is done using the function ipc_set_get_non_suspended_saps.

4.4.5 Event Handler

Figure 4.16 shows the model of a simple event handler. On an upcall from the reactor, the event handler reads data from the corresponding SAP using the function get_data. We then model computation delay in the event handler in state S2, after which the event handler outputs some data to the same SAP from which it read. Finally, the event handler returns from the upcall by communicating on the appropriate channel with its reactor.

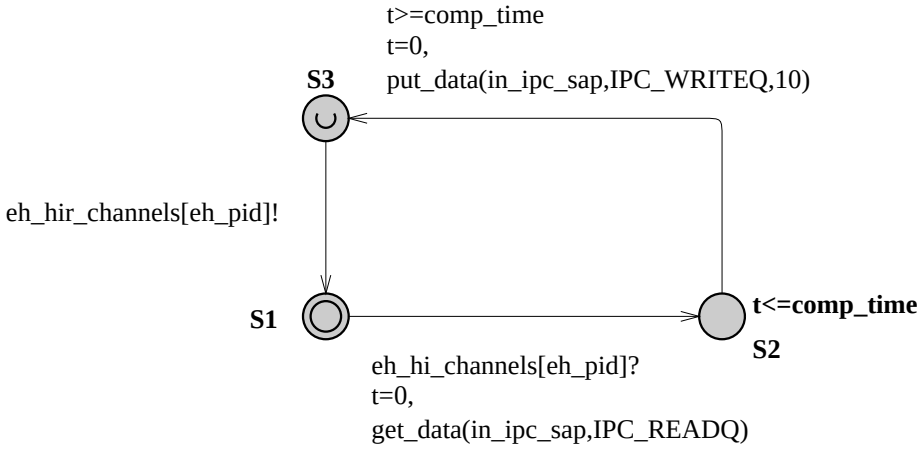


Figure 4.16: Event Handler

4.4.6 Composition of models

In this section, we give an overview of how we model a complete system by composing models of the middleware building blocks that we talked about in this chapter. We use a simple example, where a client sends a request to a server that uses a reactor and an event handler to process that request and send a reply back to the client.

First we declare a set of constants to access the different global array data structures shown in Figure 4.17. We also declare an array of Reactor_State structures to keep track of states of reactors.

```
const int REACTOR1 = 1;
......
const int EH1 = 1;
......
const int THREAD1 = 1;
const int THREAD2 = 2;
......

//States of reactors
Reactor_State reactor_states[MAX_REACTORS];
```

Figure 4.17: Composition of Models - Global Data Structures

Next, we declare a set of channels globally, as shown in Figure 4.18 so that the different automata can communicate with each other.

```
//channels for communication between automata
urgent chan handle_events_channels[MAX_THREADS];
urgent chan handle_events_return_channels[MAX_THREADS];
//Event Handler handle_input and return channels
urgent chan eh_hi_channels[MAX_FLOWS*MAX_EVENT_HANDLERS];
urgent chan eh_hir_channels[MAX_FLOWS*MAX_EVENT_HANDLERS];
//Event Handler handle_output and return channels
urgent chan eh_ho_channels[MAX_FLOWS*MAX_EVENT_HANDLERS];
urgent chan eh_hor_channels[MAX_FLOWS*MAX_EVENT_HANDLERS];

//channel used for maximal progress
urgent chan dummy;
```

Figure 4.18: Composition of Models - Channel Declarations

We then instantiate the IPC_SAP structures as shown in Figure 4.19. In this example, there are two service access points connected by means of two IPC_Channel automata. The SAP on the client side is ipc_sap_0 and the SAP on the server side is ipc_sap_1.

```
//IPC_SAP structure instantiations
IPC_SAP ipc_sap_0 = {0, false, 0};
IPC_SAP ipc_sap_1 = {1, false, 0};
```

Figure 4.19: Composition of Models - Instantiation of SAPs

The handler repository of reactors and their wait sets must be filled in appropriately as well and this is shown in Figure 4.20. In this example, ipc_sap_1 is added to the set of SAPs to be watched by the reactor and the handler repository is updated with a mapping of ipc_sap_1 to EH1.

```
int init_func()
{
  //Establish handler repository for REACTOR1
  //mappings <ipc_sap_1:EH1>
  reactor_states[REACTOR1].handler_repo.currsize = 0;
  HR_add_handler(reactor_states[REACTOR1].handler_repo,
                 ipc_sap_1,EH1);

  //REACTOR1 should watch ipc_sap_1
  ipc_set_add_member(reactor_states[REACTOR1].read_sap_set_to_watch,ipc_sap_1);

  return 0;
}
```

Figure 4.20: Composition of Models - Event Handler Registration

Now all the different middleware models are instantiated as shown in Figure 4.21. Two IPC channels are created - one modeling the connection from client1 to EH1 and the other modeling the reverse connection from EH1 to client1. An event handler is instantiated with a processing time of 10 time units and ipc_sap_1 is associated with this event handler. The event handler uses this SAP to write or read data from the global IPC channel collection. A TP_Reactor along with two threads in the thread pool is instantiated. Note that there are separate channels for the execution stack corresponding to the two threads. Finally, a client is instantiated with ipc_sap_0 as a parameter that can be used by the client to send requests to EH1.

```
//Instantiation of IPC_Channel automata
client1_eh1_fwd = IPC_Channel(ipc_sap_0,ipc_sap_1,0);
client1_eh1_rev = IPC_Channel(ipc_sap_1,ipc_sap_0,0);

//Instantiation of EventHandler automata
eh1 = EventHandler1(EH1, 10, ipc_sap_1, REACTOR1);

//Instantiation of thread pool reactors
reactor1_t1 = TP_Reactor(THREAD1,handle_events_channels[THREAD1],
                handle_events_return_channels[THREAD1],
                reactor_states[REACTOR1]);

reactor1_t2 = TP_Reactor(THREAD2,handle_events_channels[THREAD2],
                handle_events_return_channels[THREAD2],
                reactor_states[REACTOR1]);

//Instantiation of thread pools
tp_thread1 = ReactorThread(handle_events_channels[THREAD1],
               handle_events_return_channels[THREAD1]);
tp_thread2 = ReactorThread(handle_events_channels[THREAD2],
               handle_events_return_channels[THREAD2]);

//Instantiation of Client
client1 = Client(ipc_sap_0);
```

Figure 4.21: Composition of Models - Instantiation of Models

4.5 Limitations of Modeling Middleware in UPPAAL

We have seen how we developed models of middleware building blocks in UPPAAL and how to compose these models to create a model of a simple application. In the process of doing this, we have also noted some key limitations of UPPAAL, such as the lack of maximal progress semantics. While we have been able to work around the maximal progress problem, other limitations remain such as the lack of support for dynamically instantiating automata which is required to model specialized event handlers like Acceptor [92] and Connector [92] that create other event handlers during execution. However, UPPAAL supports static instantiation, which we used to instantiate the different automata during composition. Another limitation of UPPAAL is the inability to refer to specific instances of automata within our models so that we can directly specify a particular automaton with which to communicate. Instead, all communication between automata have to be made explicit using a channel, which adds a level of indirection from the perspective of a model developer. We also encountered problems collecting state space statistics in UPPAAL during exhaustive state space exploration, with the verifyta utility stopping abruptly when doing exhaustive exploration for a large state space. In Chapter 5, we show how the remaining limitations can be resolved through adopting a more sophisticated set of modeling and model checking capabilities not found in UPPAAL (*e.g.,* transition control using priority rules), which are provided by the IF toolset. In spite of all these limitations in UPPAAL, we developed our example scenarios described in Chapter 6 to demonstrate the reusability of our modeling architecture and models.

4.6 Summary

In this chapter, we have shown how UPPAAL can be used to develop and evaluate reusable timed automata models of basic middleware building blocks that are reified in the ACE [51] framework and commonly used in building distributed systems middleware. We identified modeling challenges like the absence of maximal progress semantics, and simulating two-way calls, recursive calls and thread call-stacks which

are not explicitly represented as abstractions in UPPAAL. We also presented solution techniques that we developed to address these challenges. Finally, we listed a set of limitations in UPPAAL that can be overcome by using the IF toolset for developing our models as we describe next in Chapter 5.

Chapter 5

Models in the IF Toolset

In Chapter 3, we described the different steps in building and using the middleware models that were developed in the research presented in this dissertation. In Chapter 4, we focused on realizing our modeling architecture described in Section 3.2 using UPPAAL. In this chapter, we focus on model development (step 3 in Figure 3.1) using the IF (Intermediate Format) toolset [12] [11] [10] that supports timed automata modeling and model checking.

5.1 Realization of the Middleware Modeling Architecture in IF

Figure 5.1 shows our realization of the modeling architecture discussed in Chapter 3 using the IF toolset [12] [11] [10]. Note that the generic architecture in Figure 3.2 again has been made more concrete in Figure 5.1, as it was for UPPAAL in Figure 4.1, using modeling constructs in IF. For example, the foundational data structures and operations are realized using the array, ADT, string and user-defined data-types in IF, which we discuss in Section 5.2. Property specification is done through IF observers and is discussed in Section 5.4. Transition control mechanisms are realized in IF using a combination of techniques that we describe in Section 5.5 using priority rules and observers.

We use the IF notation to specify our fine-grained models as IF *processes* that run in parallel and interact through shared variables and asynchronous signals. The behavior

of these IF processes is represented formally in IF as *timed automata with urgency* [9] and the semantics of a system modeled in IF is the Labeled Transition System (LTS) obtained by interleaving the executions of its processes.

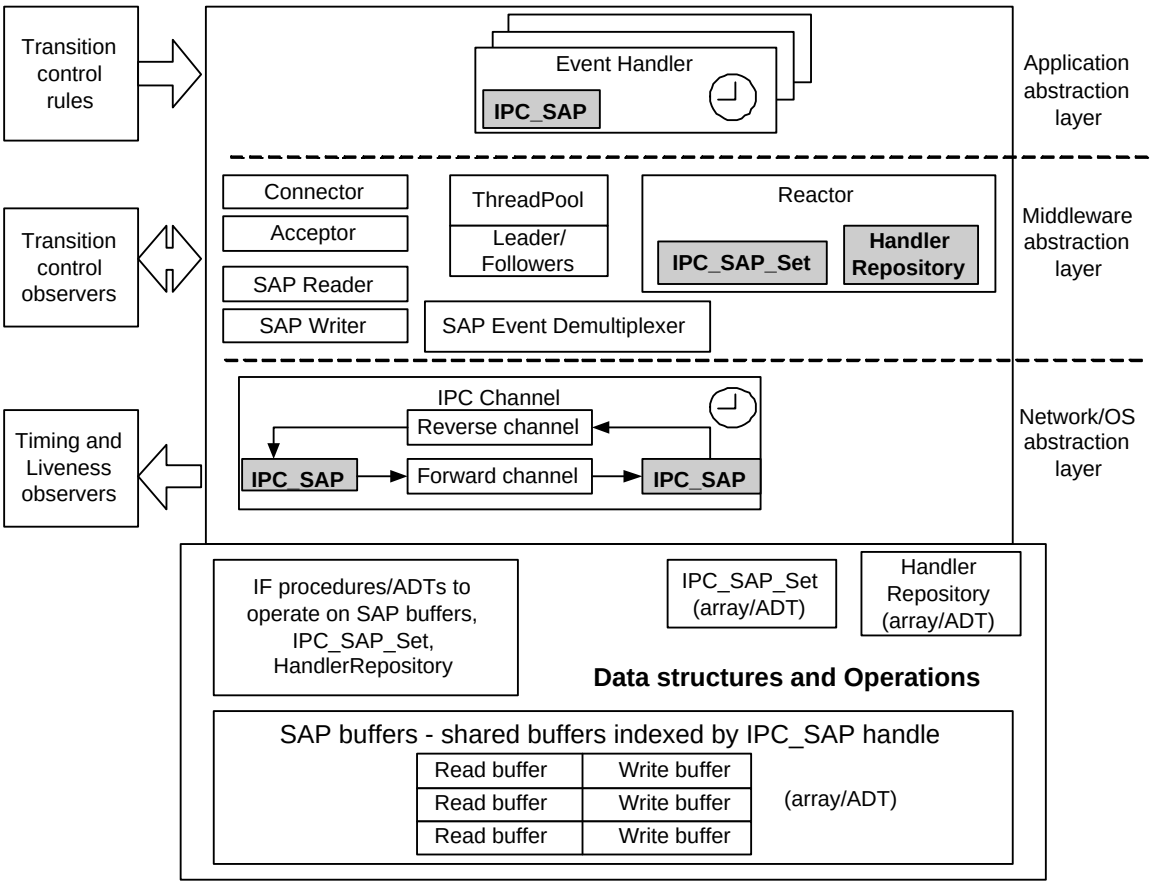


Figure 5.1: Realization of the Modeling Architecture using IF

We use a combination of the following features in IF to model the fundamental data structures and operations - Abstract Data Types, IF procedures, and arrays. We have also devised a set of strategies to reduce the state space through a set of transition control rules, using IF's priority rules and observers. The IF observers are also used to specify and verify timing and liveness properties. We now discuss each of these techniques in detail.

5.2 Modeling of Foundational Data structures and Operations

In IF, the foundational data structures and operations that we discussed in Section 3.2 can be realized using one of the following two features - (1) Abstract Data Types (ADTs) or (2) IF procedures. Both these features enable integration of external C++ code in IF to perform complex data transformations. For example, data transformation in our models includes adding/deleting items to/from the handler repository and updating the SAP buffers, and traversing arrays or other more complex data structures. These data transformation activities are much more easily expressed in a language like C++ than in the IF notation. Moreover, if they are performed within the model, such data transformation activities again increase the state space unnecessarily without affecting the outcome of verification.

ADTs in IF. Using the ADT feature in IF, one can specify the operations (interface) on an abstract data structure within the IF model, but then provide an implementation of the operations in C++ outside the IF model. The linking of each ADT with its implementation is done during the compilation of the IF model. An ADT declared without an appropriate implementation will result in a model compilation error. The implementation details of the ADT are not exposed in the IF model and therefore an ADT object is treated as a black box within the IF model thus (possibly) reducing the state space. Models in IF can then perform operations on the ADTs using the operations specified as part of the ADT declarations within the models in IF.

Procedures in IF. Whereas an ADT is treated as a black-box whose implementation is outside the IF models, the IF *procedure* feature allows one to use C++ to perform transformations on data structures declared within an IF model. These data structures can be passed as parameters to the procedure. Both call-by-value and call-by-reference semantics are supported for parameter passing and return values are also allowed. This feature differs from the ADT feature in that a procedure does not encapsulate any data of its own whereas an ADT does. This difference is similar to

the difference between a class and a function in C++. Typically, with procedures the data (on which these procedures act) is part of the state space, whereas with ADTs, the data is not part of the state space.

IF procedures act as wrappers that can perform operations on either (1) ADT implementations of data structures outside of the IF model, or (2) array-based implementations of data structures inside the IF model. We now describe the two techniques, and in Chapter 6 we compare them in terms of their effects on the state space. Intuitively, one might tend to choose the ADT implementation over the procedure implementation since the former enables the modeler to leverage powerful libraries, *e.g.,* the C++ Standard Template Library, to realize more sophisticated data structures in the IF model. We tried this initially, but realized that using the C++ STL as part of the ADT implementation could increase the state space significantly when compared to a procedure-based implementation. We show results supporting this observation in Section 6.8.1. However, for the sake of completeness, we first discuss both ADT and procedure based implementations here.

5.2.1 Foundational Operations Using IF ADTs

Figures 5.2 and 5.3 show extracts from the model of IPC SAP buffers that is shared among multiple IF processes. An IPC_SAP is declared as an IF *record* (A), and the IPC_SAP_Buffers collection is declared as an *abstract data type* (ADT) (B) in the IF model. Two IF *procedures* (C and D) are used to encapsulate this ADT based implementation. Figure 5.3 shows the ADT implementation where the two IF procedures are realized using C++ functions (G and H), which in turn access the IPC_SAP_Buffers (F) data structure realized using a C++ class. In this case, the IPC_SAP_Buffers data structure is implemented as a C++ Standard Template Library *map* (F) with elements of type IPC_SAP_Buffer (E). It should be noted that we model the read and write buffers (E) that are associated with an IPC_SAP as integers. Since the properties that we are analyzing are not influenced by the actual *contents* of the read and write buffers, it is sufficient to store only the number of bytes contained in the read and write buffers. This helps in reducing the state space of the models. Note that Figure 5.2 also illustrates how the calls to the underlying

C++ implementation (shown in Figure 5.3) to get data from (G) and put data into (H) the read and write buffers associated with a SAP, are wrapped by IF procedures (C) and (D). These can be called from inside the model to access and enqueue data for a SAP. The IPCQ_Type parameter (G and H) in the implementation of the C++ methods specifies the type of buffer (read or write). In the C++ IPC channel data structure, we access the read and write buffers for a SAP in the map using the SAP's unique handle and then increment the write buffer counter by the number of bytes to be written during a write operation (H) or decrement the read buffer counter by the number of bytes to be read during a read operation (G). Parameterizing the number of bytes to be read or written makes it easy to model the read and write OS system calls respectively. An example of this kind of usage can be seen in the IPC channel data structure that we describe in Section 5.3.1.

A

```
type IPC_SAP = record
    sap_handle integer;
    suspended integer;
    annotation integer;
endrecord;
```

B

```
type IPC_SAP_Buffers =
    abstract
        integer is_any_sap_hot(IPC_SAP_,
           IPC_SAP_Set, IPC_SAP_Set);
    endabstract;
```

C

```
procedure IPC_SAP_enqueue_data;
fpar inout sap_buffers IPC_SAP_Buffers,
       in sap IPC_SAP, in qtype IPCQType,
       in num_bytes integer;
returns integer;
{#
   int rc;
   //the following function is implemented in C++ outside IF model
   rc = sap_buffers.enqueue_data(sap, qtype, num_bytes);
   return rc;
#}
endprocedure;
```

D

```
procedure IPC_SAP_get_data;
fpar inout sap_buffers IPC_SAP_Buffers,
       in sap IPC_SAP, in qtype IPCQType,
       in num_bytes integer;
returns integer;
{#
   int rc;
   //the following function is implemented in C++ outside IF model
   rc = sap_buffers.get_data(sap, qtype, num_bytes);
   return rc;
#}
endprocedure;
```

Figure 5.2: Modeling IPC SAP Buffers with ADTs in IF

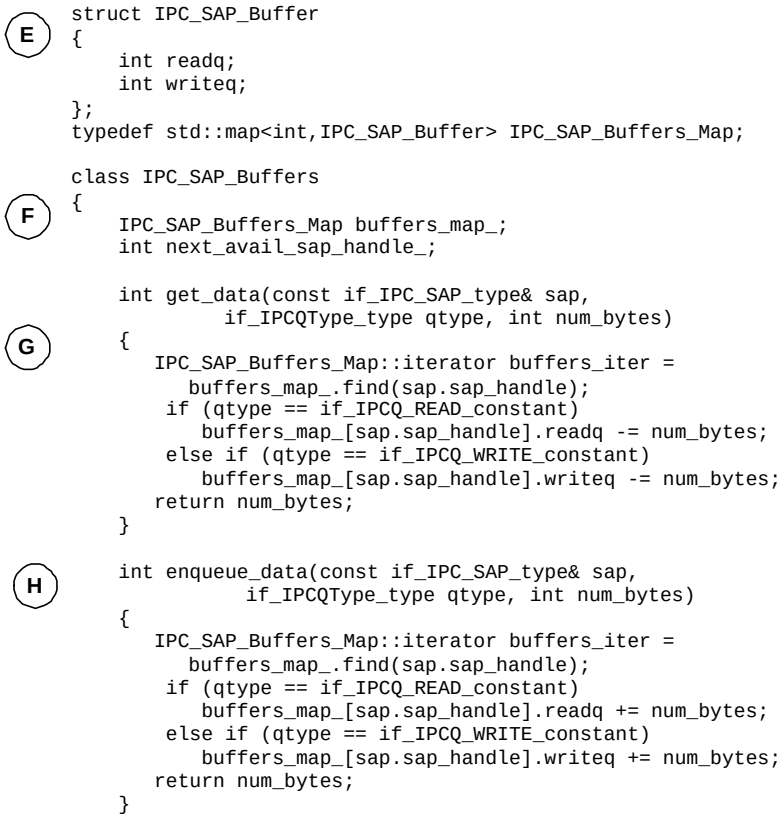


Figure 5.3: C++ implementation of IPC SAP Buffers ADT outside IF model

5.2.2 Foundational Operations Using IF procedures

Figure 5.4 shows the modeling of the foundational operations using IF procedures. In contrast with the ADT implementation, in this case all operations are implemented within the IF model. The IPC_SAP (A) data structure is declared as before. In the ADT version, the IPC_Buffer (B) data structure was part of the C++ implementation whereas here its declaration is in the IF model. The IPC_SAP_Buffers data structure (C) is declared as an array of IPC_SAP_Buffer. The procedures used to put data (D) into and get data (E) from a SAP buffer are defined within the IF model itself. The SAP handle is used to access the array of buffers. The main advantage of the ADT approach is that it makes operations like adding a new SAP or deleting an existing one easier through the usage of C++ STL data structures like vector and map. With the procedure approach, operations like deletion are cumbersome to implement using a plain C-style array.

5.2.3 Use of procedure calls as guards

When using procedures to model the foundational operations, we encountered a problem that is worth pointing out - in general some procedure cannot be used as guards in guarded transitions because those procedures could have side effects. Since each guard is evaluated at every step of the system model's execution to identify which transitions are enabled, a procedure with side effects could cause the state to change which should cause the guard to evaluate again. For example, IF does not allow the expression shown in Figure 5.5.

The problem with disallowing the above expression is that even though we have a procedure that does not have any side-effect, we cannot use that as a guard. This is a major obstacle in our case, since we monitor shared variables for identifying events in the system. For example, as we will see later in this section, the select-based reactor needs to monitor a set of SAPs to see whether any one of them has data in its buffer. If all the SAP buffers are empty, the reactor should block until one of them is ready. Since the number of SAPs to watch could vary, it is not possible to write an

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

B  type IPC_Buffer =
       record
           readq integer;
           writeq integer;
       endrecord;

C  type IPC_Buffer_Array = array[MAX_IPC_BUFFERS] of IPC_Buffer;
   type IPC_SAP_Buffers =
       record
           num_members integer;
           members IPC_Buffer_Array;
       endrecord;

D  procedure IPC_SAP_enqueue_data;
   fpar inout sap_buffers IPC_SAP_Buffers,
         in sap IPC_SAP, in qtype IPCQType, in num_bytes integer;
   returns integer;
   {#
      if (qtype == if_IPCQ_READ_constant)
         sap_buffers.members[sap.sap_handle].readq += num_bytes;
      else if (qtype == if_IPCQ_WRITE_constant)
         sap_buffers.members[sap.sap_handle].writeq += num_bytes;
      return num_bytes;
   #}
   endprocedure;

E  procedure IPC_SAP_get_data;
   fpar inout sap_buffers IPC_SAP_Buffers,
         in sap IPC_SAP, in qtype IPCQType, in num_bytes integer;
   returns integer;
   {#
      if (qtype == if_IPCQ_READ_constant)
         sap_buffers.members[sap.sap_handle].readq -= num_bytes;
      else if (qtype == if_IPCQ_WRITE_constant)
         sap_buffers.members[sap.sap_handle].writeq -= num_bytes;

      return num_bytes;
   #}
   endprocedure;
```

Figure 5.4: Modeling of IPC SAP Buffers with procedures in IF

```
//causes IF compilation error!!!!
provided (call some_procedure(params) == 1);
    //do something
```

Figure 5.5: Restrictions to Procedure Usage in IF

OR expression that includes a specific set of SAPs and hence procedures is the only option. One possible approach is shown in Figure 5.6.

```
state s0;
   result = call some_procedure(params);
   nextstate s1;
endstate;

//unstable so that s0-s1 are considered as a
//single state for execution.
state s1 #unstable ;
    provided (result = 1);
       //if condition true, do something
    provided (result = 0);
       //if condition false, check again
       nextstate s0;
```

Figure 5.6: Limitation with Usage of Procedures for Condition Wait

In this case, the condition is evaluated in one state and then the result is checked in the next state to determine whether or not the condition is satisfied. The reason for having two states is that the provided clause in IF cannot *follow* procedure call expressions. If the condition is not satisfied it is evaluated again. This is equivalent to doing a "busy wait", polling constantly until a condition is satisfied. This approach fails because the transition from s0 to s1 is always enabled and this could cause this transition to be selected by the model checker any number of times which will result in an overall state space explosion.

The solution we have devised is to use a dummy abstract data type and declare the side-effect free procedures that we want to use as guards, as operations in the dummy ADT. The IF compiler allows the use of ADT operations as guard expressions, since ADTs are treated as black boxes by IF and any side-effects will be localized within them and won't be known to IF.

The extract in Figure 5.7 illustrates our solution.Here a dummy ADT is declared with all the necessary operations and then the ADT is instantiated so that it is globally accessible from all IF processes. These ADT operations can now be used as part of provided expressions. For clarity however, in the following discussions we omit the dummy ADT parameter as part of guard expressions.

```
type Dummy =
    abstract
        integer is_cond_true(Dummy, ...);
    endabstract;
.....
process Global(0);
.....
//instantiate a globally accessible dummy ADT
Dummy dummy public;
.....
endprocess;
.....

provided is_cond_true(({ACE_Global}0).dummy, ....);
   //do something
```

Figure 5.7: Our solution to Condition Wait in IF

5.3 Modeling Middleware Building Blocks in IF

We now describe the models of middleware building blocks that we developed in IF. These models use the foundational data structures and operations that we described in Section 5.2.

5.3.1 IPC Channel

An IPC channel is bidirectional. It is modeled, however, as two data-transfer automata, one for the forward direction (from the lhs-SAP to the rhs-SAP), and one for the reverse direction (from the rhs-SAP to the lhs-SAP). The forward channel automaton waits for data to be enqueued in the write-buffer of the lhs-SAP and transfers it to the read-buffer of the rhs-SAP, as Figure 5.8 shows. The reverse channel automaton waits for data to be enqueued on the write buffer of the rhs-SAP and transfers it to the read-buffer of the lhs-SAP. These forward and reverse channel automata also can be parameterized with propagation delays, if needed. These propagation delays are implemented using transitions guarded with clock variables in the IF model. Modeling the IPC channel as an automaton with those parameters allows us to model different degrees of asynchrony and non-determinism introduced by real-world communication channels.

```
process UniDir_IPC(0);
fpar src_sap IPC_SAP,
(A) fpar dest_sap IPC_SAP,
fpar min_prop_delay integer,
fpar max_prop_delay;
……...
(B) var prop_delay clock;
……...
……...
(C) state wait_for_data_in_src_sap;
    provided is_queue_empty(({ACE_Global}0).sap_buffers,
                    src_sap_, IPCQ_WRITE) <> 1;
        set prop_delay := 0;
        nextstate data_in_src_sap;
endstate;

state data_in_src_sap ;
    deadline delayable;
(D) when prop_delay >= min_prop_delay and prop_delay <= max_prop_delay;
        ……
        call IPC_SAP_get_data(({ACE_Global}0).sap_buffers, src_sap_,
                    IPCQ_WRITE, bytes_to_transmit_);
        call IPC_SAP_enqueue_data(({ACE_Global}0).sap_buffers, dest_sap_,
                    IPCQ_READ, bytes_to_transmit_);
        nextstate wait_for_data_in_src_sap;
endstate;
```

Figure 5.8: Extracts from Channel Propagating Data between Two SAPs

The forward and reverse channel automata are instantiated when an IPC channel is created. As Figure 5.8 shows, the IPC channel uses a clock variable (A) to keep track of propagation delays. Blocking on a condition, such as an input queue being empty, is represented in the model by the IF provided clause. A forward IPC channel waits (B) on the condition that there is some data enqueued in the write buffer of its source SAP. This is implemented using the is_queue_empty ADT method. Until the condition becomes true, the IF-process modeling this channel cannot run since none of its transitions is enabled. Once that condition becomes true, the channel waits for a specified delay (C) if any, then gets the data from the write-buffer of the source SAP and puts it into the read-buffer of the destination SAP (D). Note again that we are not using actual data in the model, but instead keep track of buffer levels by incrementing and decrementing counters for the number of bytes in the buffers. In (D), notice that the automaton could transfer fewer bytes than the total present in the source write-buffer indicated by the last parameter of the call to the procedure IPC_SAP_enqueue_data. The number of bytes transmitted can be parameterized, allowing one to model the bandwidth of a channel dynamically to represent real-world phenomena like flow control.

5.3.2 Select Reactor

Each middleware primitive is modeled so that the behavior seen when the model is executed closely adheres to that of the actual implementation. This faithful modeling of the middleware primitives in turn allows high-fidelity models of higher-level middleware services, which are obtained by composing these primitive models, as we discuss in more detail in Chapters 6 and 7.

(A)
```
procedure is_any_sap_hot;
fpar in sap_buffers IPC_SAP_Buffers,
       in read_set IPC_SAP_Set,
       in write_set IPC_SAP_Set;
returns integer;
{#
   int i;
   for (i=1; i<=read_set.num_members; ++i)
   {
     if_IPC_SAP_type sap = read_set.members[i];
     if (sap_buffers.members[sap.sap_handle].readq > 0)
        return 1;
   }

   //similarly for write set
   ……...
   return 0;
#}
endprocedure;
```

(B)
```
procedure get_hot_read_saps;
fpar in sap_buffers IPC_SAP_Buffers, in sap_read_set IPC_SAP_Set;
returns IPC_SAP_Set;
{#
   for (i=1; i<=read_set.num_members; ++i)
   {
     if_IPC_SAP_type sap = read_set.members[i];

     if (sap_buffers.members[sap.sap_handle].readq > 0)
        add_sap(hot_sap_set, sap);
   }
  return hot_sap_set;
#}
endprocedure;
```

(C)
```
procedure get_hot_write_saps;
fpar in sap_buffers IPC_SAP_Buffers, in sap_write_set IPC_SAP_Set;
returns IPC_SAP_Set;
{#
   for (i=1; i<=write_set.num_members; ++i)
   {
     if_IPC_SAP_type sap = write_set.members[i];

     if (sap_buffers.members[sap.sap_handle].writeq < MAX_BUF_SIZE)
        add_sap(hot_sap_set, sap);
   }
  return hot_sap_set;
#}
endprocedure;
```

Figure 5.9: Extracts of Foundational Operations used by Select Reactor Model

```
state start_listen ;
    provided is_any_sap_hot(({ACE_Global}0).sap_buffers, sap_read_set_,
(A) sap_write_set_) = 1;
        hot_saps_read_set_ :=
       call get_hot_read_saps(({ACE_Global}0).sap_buffers, sap_read_set_);
        hot_saps_write_set_ :=
       call get_hot_write_saps(({ACE_Global}0).sap_buffers, sap_write_set_);
        nextstate dispatch_event_handlers;
….......
…….
state dispatch_event_handlers;
(B)  provided size(hot_saps_read_set_) > 0;
        next_hot_sap := call pop_first_SAP(hot_saps_read_set_);
        event_handler := call get_handler(({Reactor}reactor_).handler_rep_,
next_hot_sap);
        output handle_input(context, next_hot_sap) to event_handler;
        nextstate wait_for_handle_input_return;

(C)  provided size(hot_saps_read_set_) <= 0 and size(hot_saps_write_set_) > 0;
        next_hot_sap := call pop_first_SAP(hot_saps_write_set_);
        event_handler := call get_handler(({Reactor}reactor_).handler_rep_,
next_hot_sap);
        output handle_output(context, next_hot_sap) to event_handler;
        nextstate wait_for_handle_output_return;
```

Figure 5.10: Extracts from the IF Based Model for Select Reactor

Our select-based reactor model illustrated in Figures 5.9 and 5.10 uses the foundational data structures and associated operations to query the I/O status of different SAPs - *e.g.,* whether data is ready to be read or written. As shown in Figure 5.9, the is_any_sap_hot procedure is used to query whether any SAP from among a set of SAPs is *ready* - *i.e.*, whether an I/O event can be performed on that SAP without blocking. IF procedure is_any_sap_hot (Figure 5.9 A) is used to determine whether any SAP is ready to be read from or written to without blocking. IF procedures get_hot_read_saps ((Figure 5.9 B) and get_hot_write_saps ((Figure 5.9 C) are used to obtain the read-ready and write-ready SAPs respectively. As shown in Figure 5.10, the Select_Reactor_Handle_Events automaton calls (A) the is_any_sap_hot procedure to wait for I/O events on a set of SAPs. Once the event arrives its guard condition becomes true and it then calls get_hot_read_saps (B) and get_hot_write_saps to obtain the read-ready and write-ready SAPs respectively. For each of the "hot" SAPs, the handler repository is accessed using the IF procedure get_handler to obtain a reference to the corresponding event handler. The handle_input (Figure 5.10 B) or handle_output (Figure 5.10 C) IF signal is then sent to the event handler, depending on whether the

SAP is ready for reading or writing respectively. Sending these signals models the the initiation of the respective handle_input and handle_output upcall dispatch actions performed by the actual select-based reactor implementation in ACE.

5.3.3 Thread Pool Reactor

```
state dispatch_event_handlers;
    provided size(hot_saps_read_set_) > 0;
        next_non_suspended_hot_sap :=
A    call pop_first_non_suspended_sap(hot_saps_read_set_);
        event_handler :=
B    call get_handler(({Reactor}reactor_).handler_rep_,
                next_non_suspended_hot_sap);
C       call suspend_sap(({Reactor}reactor_).sap_read_set_,
                   next_non_suspended_hot_sap);
        output handle_input(context,
D                 next_non_suspended_hot_sap) to event_handler;
E       task ({Reactor}reactor_).handle_events_in_progress_ :=
    ({Reactor}reactor_).handle_events_in_progress_ - 1;
        task suspended_sap_ := next_non_suspended_hot_sap;
        nextstate wait_for_handle_input_return;

state wait_for_handle_input_return;
F   input handle_input_return(par_context, rc);
            call resume_sap(({Reactor}reactor_).sap_read_set_,
G                   suspended_sap_);
        endif

        nextstate done;
endstate;
```

Figure 5.11: Extracts from Thread Pool Reactor Model

Figure 5.11 shows parts of the IF model for the ACE [51] Thread Pool (TP) reactor. It should be noted that there is a distinct difference between a single-threaded reactor model illustrated in Figure 5.10 and the thread pool reactor model shown in Figure 5.11. In the former, a single thread uses the reactor to wait on multiple I/O channels. Once a set of SAPs becomes ready, this thread iterates through the set of ready SAPs and dispatches upcalls to each of the corresponding event handlers sequentially. Only after *all* upcalls have been dispatched, does the thread return to the reactor to watch for I/O events again. In contrast, in the TP reactor model shown

in Figure 5.11, a leader thread chooses a non-suspended SAP from among the ready SAPs. A call to the IF procedure pop_first_non_suspended_sap is made (A) to extract this information from the set of ready SAPs. The leader thread then obtains (B) the corresponding event handler using the get_handler IF procedure. It suspends (C) this SAP using the IF procedure suspend_sap before making the upcall (D). The leader thread then waits for the upcall to be completed (F). On completion of the upcall, the leader thread resumes (G) the suspended SAP. Note that the IF procedures take the reactor's SAP set as an inout parameter and hence all modifications made in the C++ code are reflected in the SAP set owned by the reactor.

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

5.3.4 Event Handler

For example, Figure 5.12 shows how an event (service) handler is modeled in both a time driven and an event driven manner. The event handler waits (A) for an upcall event from its associated reactor. In this example, the event handler gets an event indicating that the SAP associated with the event handler is ready to be read. Once it gets the handle_input IF signal from the reactor, the event handler reads data (B) from the SAP using the IF procedure IPC_SAP_get_data. After the read, the event handler spends 10 time units on its computation (C), which is modeled by means of the "when" IF clause. The "deadline delayable" qualifier indicates that the transition should be delayed until 10 units of time has elapsed. After that, the event handler sends a request to another event handler using a SAP (D). The sending of the request is modeled by writing (D) to the appropriate SAP corresponding to the input SAP of the destination event handler, using the IF procedure IPC_SAP_enqueue_data.


```
process Service_Handler(0);
var Reactor_ reactor;
state wait_for_reactor_upcall;
  input handle_input(par_context, par_sap);
    nextstate do_read;
endstate;

state do_read;
    call IPC_SAP_get_data(({ACE_Global}0).sap_buffers,
                          in_sap_, IPCQ_READ);
    nextstate do_compute;
endstate;

state do_compute;
  deadline delayable;
  when elapsed = 10;
    //send a request to to another service handler
    call IPC_SAP_enqueue_data(({ACE_Global}0).sap_buffers,
                          out_sap_, IPCQ_WRITE);
    nextstate wait_for_reply;
endstate;

state wait_for_reply;
  provided is_queue_empty(({ACE_Global}0).sap_buffers,
                          out_sap_, IPCQ_READ) <> 1;
    call IPC_SAP_get_data(({ACE_Global}0).sap_buffers,
                          out_sap_, IPCQ_READ);
……….
endstate;
```

Figure 5.12: Extracts from the Model of a Service Handler

5.4 Property specifications for verification

In the IF toolset, properties to be checked can be specified by observers [12]. These observers are also represented by timed automata and are executed at each step of the labeled transition system (LTS) that is generated from the composed system model before an enabled transition is selected. To facilitate specification, IF provides observer constructs for a variety of events in a system including forking a new process, output events, and input events. In general an observer records an abstraction of the actions and interactions of other automata. The observer can also be used to control the extent of the state space that is explored through the cut statement, which cuts off selected execution paths. An observer can also be intrusive and act as an *interceptor* [97] to change the system state by modifying variables or sending signals. Examples of such observers are shown in Section 5.5, where we use an intrusive observer to help reduce the state space.

We use IF *observers* [12] to specify system properties for verification. Observers can be used in conjunction with timed transitions to detect deadline misses in the model, as we described in Section 3.1. Figure 5.13 illustrates part of a writer process that sends (A) a request to another process and then waits for the reply. If the reply (B) does not arrive before a particular deadline, then there is a deadline miss (C) which we need to verify. The observer for the deadline miss can be written using the IF instate primitive, which triggers an observer transition when the specified process is in the specified state, *e.g.,* a state representing a deadline miss, or a state at which the clock value should be checked. The observer may also perform internal state transitions, enabling us to capture state within the observer itself. The observer illustrated in Figure 5.13 starts monitoring events (D) when the main process ($\{Main\}0$) in the system has completed system initialization and reaches the *done* state. The observer then keeps monitoring the state of the two writer processes. If any one of the writer processes misses its deadline, indicated by state err (E), then there was a deadline miss and the observer moves to an error state (G). When both of the writer processes are in their done state (F), then that means that there were no deadline misses. In this case, the observer moves to a success state (H).

```
process WriterProc(0);
var elapsed clock;
state do_stuff;
A       call IPC_SAP_enqueue_data(({ACE_Global}0).sap_buffers,
                    sap_, IPCQ_WRITE);
        set elapsed := 0;
        nextstate listen;
endstate;
state listen;
B   provided is_queue_empty(({ACE_Global}0).sap_buffers, sap_,
                                    IPCQ_READ) <> 1;
        call IPC_SAP_get_data(({ACE_Global}0).sap_buffers, sap_,
                            IPCQ_READ);
C       nextstate done;
    when elapsed > rel_deadline_;
        nextstate err;
endstate;
endprocess;

cut observer deadline_miss_observer;
state init #start ;
    deadline eager;
D   provided ({Main}0) instate done;
        nextstate startmonitor;
endstate;
state startmonitor;
E   provided ({WriterProc}0) instate err or
              ({WriterProc}1) instate err;
        nextstate err;
F   provided ({WriterProc}0) instate done and
              ({WriterProc}1) instate done;
        nextstate succ;
endstate;
G state err #error ;
    cut;
    nextstate -;
endstate;
H state succ #success ;
    cut;
    nextstate -;
endstate;
endobserver;
```

Figure 5.13: Cut Observer Based System Property Specifications

5.5 Issues for Modeling Concurrent Object-Oriented Systems in IF

As we noted earlier, the primitive mechanism for modeling behavior in IF is a *process*. Although our goal is to model systems having multiple communicating threads, each of which executes actions including object method calls, the distinction between an object and a thread is not known to the IF model checker. Essentially, we need to express an object-oriented concurrent communicating system in terms of a process calculus. This makes it the responsibility of the IF model developer to keep track of this distinction in the model itself, which contributes in part to the state space. For example, each object method call has to be simulated by forking a new IF process representing the method call.

This is necessary especially in situations like the *WaitOnReactor* [103] strategy for waiting for replies which involves recursive calls to the handle_events method that is invoked on the reactor to wait for multiple I/O events occurring on different SAPs. Such situations motivate the need for modeling the handle_events method call's execution using a separate IF process. Since the behavior of handle_events is different in the case of TP_Reactor and Select_Reactor we have two distinct IF processes for these reactors' handle_events methods. The reactor itself is represented using a single IF-process and it can be shared by multiple threads. Thus the reactor forks an IF process, if necessary, to model a method call in the reactor in the context of a thread. Two distinct automata (TP_Reactor_Handle_Events and Select_Reactor_Handle_Events) are used in our model to simulate the behavior of the handle_events method in the corresponding ACE Reactor implementations.

5.5.1 Modeling Object Interactions in IF

We model both objects and threads using IF processes. For example, a method call on an object such as an event handler, made from another object such as a reactor, is modeled by the caller object process sending an IF signal to the callee object process. The caller process then waits for a reply from the callee process. The reply is also modeled as an IF signal, from the callee process back to the caller process.

Modeling an object as a single IF process has certain drawbacks. It is difficult to model concurrency in such a model since there could be multiple threads calling methods of an object. If we model an object using a single IF-process and the automaton is in the middle of executing a method, then it may not be ready to receive any other method invocations. This enforces an implicit serialization among threads accessing an object, which is sufficient for modeling synchronized middleware primitives like monitor objects [97]. However, to accommodate complex behavioral modeling within an object method, our approach if to fork a new IF-process for each method invocation that has a relatively complex automaton (involving multiple states) [38], as in the case of the handle_events method of the reactor.

This notion of modeling an object as a process in IF rather than having a native construct for an object can be considered a drawback of IF for our particular purpose of modeling distributed real-time embedded middleware. At the same time, however, it should be noted that the original intention of IF is to model communicating processes rather than object systems. The specific limitation for our purposes is that there is no native concept of an object or thread in IF. Bogor [86] was designed to support object-oriented systems, and has constructs that can express objects and threads directly in its modeling language, but does not support timed automata.

5.5.2 Modeling Threads in IF

Since as we noted earlier IF does not have distinct native support for constructs like threads and objects, it is the responsibility of the model developer to represent explicitly, in the model itself, the idea of a thread of control flowing through multiple objects as part of a chain of object method invocations. To model a distinct thread flow of control, we developed the concept of a *thread id* maintained by each IF process. The *thread id* is a reference (of type pid in IF) to a unique instance of an IF process of type Thread. Note that the Thread automaton has been developed as part of this research and it is not an in-built feature in IF. This Thread automaton serves to record the real-world thread context under which that IF process is executing.

As a convention, any thread in the real-world should be modeled by creating a Thread process in our models. When we model an object method invocation, the thread

context (represented by a unique thread id) under which that invocation is made must be carried over from the caller to the callee object. To propagate the thread context, we developed an IF intrusive observer, excerpts from which are shown in Figure 5.14.

Between any two *labeled transition system* (LTS) steps, this observer runs and updates the thread context of a destination process of an IF signal to be the same as the thread context of the source process. The observer observes the output of a signal from a source process to a destination process and updates the thread context of the destination process to be the same as the source process. This approach has the disadvantage that if a process sends multiple signals to other processes, then it will be akin to having two concurrent flows of control within the same thread. In our case, however, we use this technique for modeling thread context propagation along object method invocations, where each method invocation has only one destination.

5.5.3 Modeling Priority Based Thread Scheduling in IF

In our model, a Thread can be instantiated with a priority that can be used to restrict non-determinism in the model just as a thread priority (*e.g.,* POSIX priority) is used as a mechanism in OS thread scheduling to reduce the number of possible interleavings of CPU instructions among threads. Each instance of the Thread automaton has a priority associated with it. Since the Thread automaton is not in-built construct in IF, there should be some mechanism by which we inform the model checker about the priorities of threads so that the model checker gives preference to transitions that are executing under the context of a higher priority thread than a lower priority thread. We use the priority rules feature in IF to specify the priority ordering among different threads.

When doing model exploration, the IF state space explorator selects one transition non-deterministically from a set of enabled transitions. To reduce non-determinism, IF priority rules can be used by the modeler to specify a preference among the IF processes that have at least one enabled transition. Preference can be specified as a set of rules, each of which expresses the ordering of two specific IF processes. For

```
intrusive observer ThreadId_Propagator;

var pid1 pid;
var pid2 pid;
var context Context;
var so t_if_signal;

state start_act;
    match fork(pid2) in pid1;
        //when a Reactor automaton forks a TP_Reactor_Handle_Events
        //automaton instance, propagate the threadid from the former
        //to the latter.
        if pid1 instanceof Reactor and
           pid2 instanceof TP_Reactor_Handle_Events then
            task ({TP_Reactor_Handle_Events}pid2).threadid_ :=
                 ({Reactor}pid1).threadid_;
        endif
        //when a Reactor automaton forks a Select_Reactor_Handle_Events
        //automaton instance, propagate the threadid from the former
        //to the latter.
        if pid1 instanceof Reactor and
           pid2 instanceof Select_Reactor_Handle_Events then
            task ({TP_Reactor_Handle_Events}pid2).threadid_ :=
                 ({Reactor}pid1).threadid_;
        endif
        nextstate -;

    match output(so) from pid1 to pid2;
        //when a Reactor automaton sends an IF signal (method call)
        //to a Select_Reactor_Handle_Events automaton instance,
        //propagate the threadid from the former to the latter.
        if pid1 instanceof Reactor and
           pid2 instanceof Select_Reactor_Handle_Events then
            if obs_queue_length(pid2) > 0 and
               obs_queue_get_first(pid2) = so then
                task ({Select_Reactor_Handle_Events}pid2).threadid_ :=
                     ({Reactor}pid1).threadid_;
            endif
        endif
```

Figure 5.14: IF Observer to Propagate Threadid

example, Figure 5.15 shows how we use the priority rules feature to achieve thread scheduling in our models.

```
scheduler_prio: pid1 < pid2

  if
      //Rule1
      ( pid1 instanceof Reactor and pid2 instanceof Reactor and
      ({Reactor}pid1).threadid_ <> ({Reactor}pid2).threadid_ and
      ({Thread}(({Reactor}pid1).threadid_)).prio <
          ({Thread}(({Reactor}pid2).threadid_)).prio ) or

      //Rule2
      ( pid1 instanceof Reactor and pid2 instanceof Select_Reactor_Handle_Events
      and
      ({Reactor}pid1).threadid_ <> ({Select_Reactor_Handle_Events}pid2).threadid_
      and
      ({Thread}(({Reactor}pid1).threadid_)).prio <
          ({Thread}(({Select_Reactor_Handle_Events}pid2).threadid_)).prio ) or
      ...
      ...
```

Figure 5.15: IF Priority Rules to Model Thread Scheduling

Rule1 states that between two automata each an instance of the IF process type Reactor, if the thread contexts of these two automata are different, then IF should choose the automaton whose thread priority is higher. To access an instance variable (*e.g.,* threadid_) within an automaton in IF, we should use typed expressions like {Reactor}pid1, {Reactor}pid2, *etc.* where a pid reference to a IF process is prefixed by the appropriate type. To avoid run-time errors, the pid reference (pid1, pid2, *etc.*) should be checked for the correct type before performing a "dynamic cast" to the appropriate type by using the instanceof operator in IF. Priority rules can also be expressed among different process classes, for example, between instances of an IF-process P and instances of another IF-process Q (Rule2 is an example of this).

First, we access the thread context by using the threadid_ instance variable, which is present in every automaton in our model. This gives us a pid reference to the Thread automaton representing that thread context. Next, the priority of a Thread automaton instance can be obtained by accessing the prio instance variable, which is populated at Thread instance creation time. Therefore these rules inform the model checker as to which process to select from a set of processes, each with at least one enabled transition. The above rule is repeated for all pairs of process types in our

model because using a priority rule we can specify the ordering between only *two* IF processes. Since these rules follow a specific syntax, we have developed tools that generate the whole set of rules, taking as input the names of the different process types in our model.

5.5.4 Modeling Run-to-Completion Semantics

In the previous section, we discussed how to model priority based scheduling in IF, where we dealt mainly with modeling threads with *different* priorities. In real-time operating systems, it is very common to use the SCHED_FIFO scheduling mechanism to control preemption between real-time threads of the *same* priority. We use a similar technique in our models to control interleaving between the model execution of two threads with the same priority. We model SCHED_FIFO semantics to control

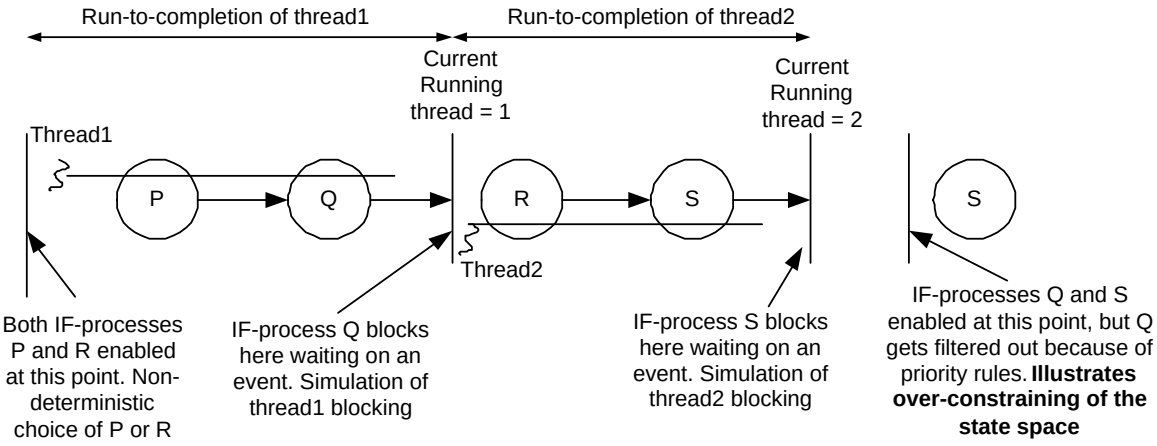


Figure 5.16: Run-to-Completion Semantics for Two Threads

unnecessary interleavings. Each logical thread of control will run across multiple IF processes until the thread blocks, and only then can another thread of control start running. Figure 5.18 illustrates this situation for two threads of control. Each of these threads passes through multiple objects. Thread1 flows through objects P and Q and Thread2 flows through objects R and S. Thread1 runs to completion before Thread2 can run, where completion means that a thread completes a phase of its activity that is totally CPU bound. In the IF model, this translates to the notion of

processes in the same thread context executing in sequence until there are no enabled transitions in the group of processes running under that thread context.

To realize the run-to-completion semantics in IF, we developed a combination of techniques: (1) keeping track of the currently running thread id as part of the state space; (2) performing thread context propagation from a caller object method invocation to callee object method invocation (two threads could be in the same object or even the same method but not in the same method invocation at once); and (3) using an idle catcher to reset the currently running thread when none of the processes in our model have any enabled transitions.

Currently running thread context. Each transition in every process in the model updates the globally accessible state variable ({Scheduler}0).current to record the thread context under which it is currently running. Any IF process P whose thread context is the same as the currently running thread will get preference to any IF process Q whose thread context is not the same as the currently running thread, provided the threads for P and Q have the same priority. This policy can be expressed in IF using a combination of IF priority rules.

```
run_to_completion: pid1 < pid2

  if
      ( pid1 instanceof Reactor and pid2 instanceof Reactor and
      ({Reactor}pid1).threadid_ <> ({Reactor}pid2).threadid_ and
      ({Thread}(({Reactor}pid1).threadid_)).prio =
          ({Thread}(({Reactor}pid2).threadid_)).prio and
      ({Scheduler}0).current = ({Reactor}pid2).threadid_ )  or

      ( pid1 instanceof Reactor and pid2 instanceof Select_Reactor_Handle_Events
      and
      ({Reactor}pid1).threadid_ <> ({Select_Reactor_Handle_Events}pid2).threadid_
      and
      ({Thread}(({Reactor}pid1).threadid_)).prio =
          ({Thread}(({Select_Reactor_Handle_Events}pid2).threadid_)).prio and
      ({Scheduler}0).current = ({Select_Reactor_Handle_Events}pid2).threadid_ )  or
      ....
```

Figure 5.17: Priority Rules to Achieve Run-to-completion Semantics

The extract in Figure 5.17 shows an instance of this *run_to_completion* rule. The semantics of this IF rule is that for any two processes P and Q, Q has preference if their thread contexts are different, their thread contexts have the same priority

and Q has a thread context that is the same as the context of the currently running thread whose id is stored in a globally accessible state variable ({Scheduler}0).current. Note that if there is no currently running thread context, then we do allow non-determinism in the system. This is because the boolean condition in the priority rules that evaluates whether a process' thread context is the same as the currently running thread's context, returns false for each process and hence there is no preference for any particular process under the run-to-completion priority rules.

Thread context propagation. To realize run-to-completion semantics in IF, it is not sufficient that a globally accessible state variable is updated with the current thread id at the beginning of the action section associated with every transition. This is because the priority rules in IF are executed in the context of the current global state of the system. The state of a process thus must be updated with the thread context under which it is running, *before* the execution of priority rules, since this is the state that is used by the priority rules. By updating the current thread at the beginning of a transition, this update happens *after* a decision is made in the model checker as to which process to execute next from among the list of processes with enabled transitions. This update is necessary, however, to allow non-determinism in the system, and consequently whichever process runs first runs to completion. The thread context propagation that we discussed earlier propagates the thread context *before* the execution of priority rules since the propagation is done by an observer that executes *before* any IF process and evaluation of priority rules.

Idle catcher. The combination of the two previous techniques is sufficient as long as there is always an enabled transition in the system. However, there could be problems when there are no enabled transitions in the system, for example when time needs to progress in the model. Figure 5.18 illustrates such a problem, where Thread1 (at process P) and Thread2 (at process R) are enabled at (1), where a non-deterministic choice is made between P and R. Assuming that process P is selected to run by the model checker, Thread1 blocks when process Q blocks at (2) waiting for some event. Process R is then selected to run and Thread2 runs to completion at (3). Note that the current running thread is updated at (1) and (2) to be Thread1 and then Thread2 respectively. At (3) Thread2 blocks, when S blocks waiting on some event. At (3), the current thread is still recorded as being Thread2. As a consequence, at (4) when

Q and S are both enabled, only S is selected by the model checker since the current thread is recorded as being Thread2. This results in over-constraining the state space, in which a form of non-determinism which is quite possible and which may be relevant to the constraints of the actual system that we are trying to model, is removed. To avoid such over-constraining, we add an Idle_Catcher process as Figure 5.18 shows. This process has a lower preference than any other process in the model, and runs

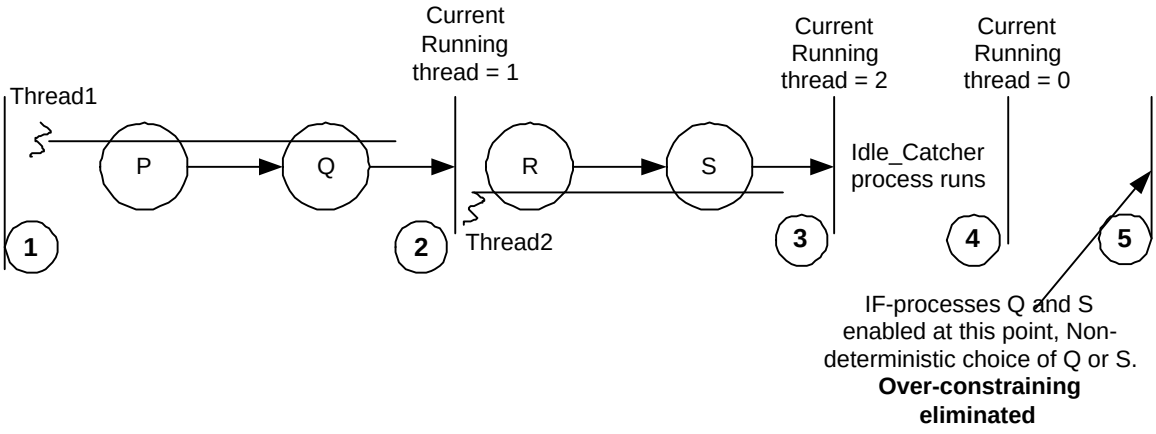


Figure 5.18: Idle Catcher

only when there are no other enabled transitions in the system (3). As soon as it runs, it resets the state variable that records the currently running thread (4). Now, when Q and S are enabled (5) one of them is picked non-deterministically by the model checker. The selected process then updates the currently running thread context and runs to completion.

5.5.5 Ordering Optimizations

A common problem with model-checking is that the state space to be checked can become intractably large. Hence, it is very important to prune out unnecessary non-determinism in the model, and avoid state transitions that do not have a representation in the real world, so that model-checking becomes more tractable. In this section, we describe a combination of techniques we have employed to achieve such state space reductions. We are particularly concerned with techniques that can

exploit information that is specific to a given application, such as its function call graph or details about its run-time environment.

Because our models allow different interleavings of actions, particularly when the models represent multiple threads of execution, it is important to distinguish interleavings of actions that are relevant to the application constraints, from spurious interleavings that could easily render the model's state space intractable. We first examine interleavings caused by the order in which objects are initialized or the order in which threads waiting for a synchronization token are chosen.

System initialization. When we construct the model of a system, we first establish the static structure of the system, creating both active and passive objects, and establish the associations between them. In IF, we use the IF process construct to model both active and passive objects. For example, a reactor is a passive object and a thread pool is an active object. Even though a reactor is a passive object, since it is modeled as an IF process, within the model checker the reactor is represented as an automaton that can run concurrently with other automata. During the static initialization phase, it does not really matter in what order the different objects and their associations are created. In other words, the different orders of those initializations are *observationally equivalent*. However, unless this information is conveyed to the model checker, it may explore permutations of possible interleavings of actions that result in the same application semantics. For example, an application may have an object A that creates an instance of object C and also may have another object B that creates another instance of C, but it does not matter to the application which instance of C is created first. However, unless told to do otherwise, the IF model checker may still explore which instance of C gets created first. In IF, when a process C is forked to model object creation, it is given a unique id. The first instance of the process can be accessed as $\{C\}0$, the second instance as $\{C\}1$ and so on. If a process $\{A\}0$ forks a process C before another process $\{B\}0$, then the instance used by $\{A\}0$ is $\{C\}0$ and the instance used by $\{B\}0$ is $\{C\}1$. On the other hand, if $\{B\}0$ forks C before $\{A\}0$ then $\{B\}0$ owns $\{C\}0$ and $\{A\}0$ owns $\{C\}1$. These are different scenarios as seen by the model checker, and these application-irrelevant interleavings during the initial phase of system structure could significantly impact the size of the state space.

By constraining such interleavings, therefore we can help reduce the overall state space. We use priority rules again to control such interleavings. During system initialization, among IF processes of the same process class, an arbitrarily selected fixed order - *e.g.,* ascending order of pid values - can be specified by the model developer for resolving such non-determinism. The priority rule named init_rules in Figure 5.19 is an example of a rule that controls interleavings during system initialization.

```
priorityrules;
init_rules: pid1 < pid2
 if
   ({Global}0).run_mode = MODE_INIT) and
   (pid1 instanceof ReaderProc and pid2 instanceof ReaderProc and
     {integer}pid1 > {integer}pid2) or (pid1 instanceof WriterProc and
    pid2 instanceof WriterProc and {integer}pid1 > {integer}pid2)
endpriorityrules;
```

Figure 5.19: Initialization Mode Priority Rule in IF

The rule in Figure 5.19 states that during system initialization, among two processes of type ReaderProc (and similarly for WriterProc), one with a lower integer valued pid would get preference. An IF-process P when instantiated multiple times would have pids as $\{P\}0$, $\{P\}1$, *etc.* If $\{P\}i$ is the pid of an IF-process P, then i is the integer value of the pid. For example, if both $\{ReaderProc\}0$ and $\{ReaderProc\}1$ have enabled transitions, then because of this priority rule, $\{ReaderProc\}0$ gets preference and hence the number of interleavings is reduced.

Leader thread election. With some concurrency strategies, such as the thread pool reactor described in Section 5.3.3, it may not matter in which order a thread is chosen from a set of waiting threads, *e.g.,* to become the leader thread to start waiting for events on the reactor. If the choice of a specific thread does not have any consequences for the safety, timing, or liveness properties of the system, then this non-determinism can be eliminated, thus reducing the state space. We use a simple strategy to remove non-determinism in this case: among the IF processes representing the waiting threads, we choose the one with the lowest process id number. The extract in Figure 5.20 shows the priority rule that enforces this.

```
leader_thread_election: pid1 < pid2
    if  (
          pid1 instanceof TP_Reactor_Handle_Events and
          pid2 instanceof TP_Reactor_Handle_Events and
          ({TP_Reactor_Handle_Events}pid1).reactor_ =
               ({TP_Reactor_Handle_Events}pid2).reactor_ and
          ({Reactor}({TP_Reactor_Handle_Events}pid1).reactor_).
                          handle_events_in_progress_ = 0 and
          ({Reactor}({TP_Reactor_Handle_Events}pid1).reactor_).
                          handle_events_thread_ <> pid1 and
          ({Reactor}({TP_Reactor_Handle_Events}pid1).reactor_).
                          handle_events_thread_ <> pid2 and
          {integer}pid1 > {integer}pid2
        );
```

Figure 5.20: Priority Rule in IF for Leader/Followers ThreadPool

5.6 Summary

In this chapter, we have shown how we used IF to develop reusable timed automata models of basic middleware building blocks that are reified in the ACE [51] framework and are commonly used in building distributed systems middleware. We have seen how automata (*i.e.*, IF processes) can be instantiated at model execution time using the fork construct in IF. We discussed various techniques that we developed for state space optimization as well as to model object-oriented concurrent systems in IF. In the next chapter we show our use of the models we developed in the context of example application scenarios that demonstrates the benefits of using our models.

Chapter 6

Representative examples

In chapters 4 and 5, we discussed the executable models of middleware building blocks that we have developed using UPPAAL and IF. In this chapter, we develop both UPPAAL and IF models of some representative scenarios that use variations of reactor, event handler and thread pool configurations. The behaviors of these example scenarios are described using informal analysis based on domain knowledge. The examples that we describe here are simple examples used for the purpose of illustrating modeling using our models described in Chapters 4 and 5 and establishing basic concepts for the discussion of model validation in chapter 7. We will present more sophisticated examples in the DA reactor case study in chapter 8, and the gateway example case study in chapter 9. Although the scenarios in this chapter are simple in concept, they serve to capture key interference issues caused by middleware building blocks. Finally we present statistics on the costs of checking these models and arrive at some crucial conclusions about necessary optimizations based on these statistics.

We choose scenarios that capture the key behavioral characteristics of the elements of our computational model discussed in Section 3.1. In this chapter, we examine these scenarios informally using our domain knowledge and then support our examination with traces from executing models of these scenarios. In the next chapter, we offer a more detailed analysis based on these scenarios to check the fidelity of our IF models against experimental observations. The purpose of developing the models for these scenarios is twofold - (1) it allows us to debug our models using informal knowledge that we have about the behavior of the modeled middleware in these scenarios, and (2) it serves to evaluate key mechanisms that we use to realize our middleware modeling

architecture. We develop our models using both UPPAAL and IF, and show that IF provides a better facility for obtaining execution traces and post-processing them.

6.1 Experimental Setup

Figure 6.1 shows the experimental setup that we used to model the different scenarios in both IF and UPPAAL. This setup served two purposes - (1) it enabled reuse of parameterized UPPAAL/IF models across different scenarios and thus helped us in writing drivers that can be used to configure these parameterized models for the different scenarios; (2) it enabled consistent terminology for modeling in both IF and UPPAAL which is important when analyzing execution traces from model execution.

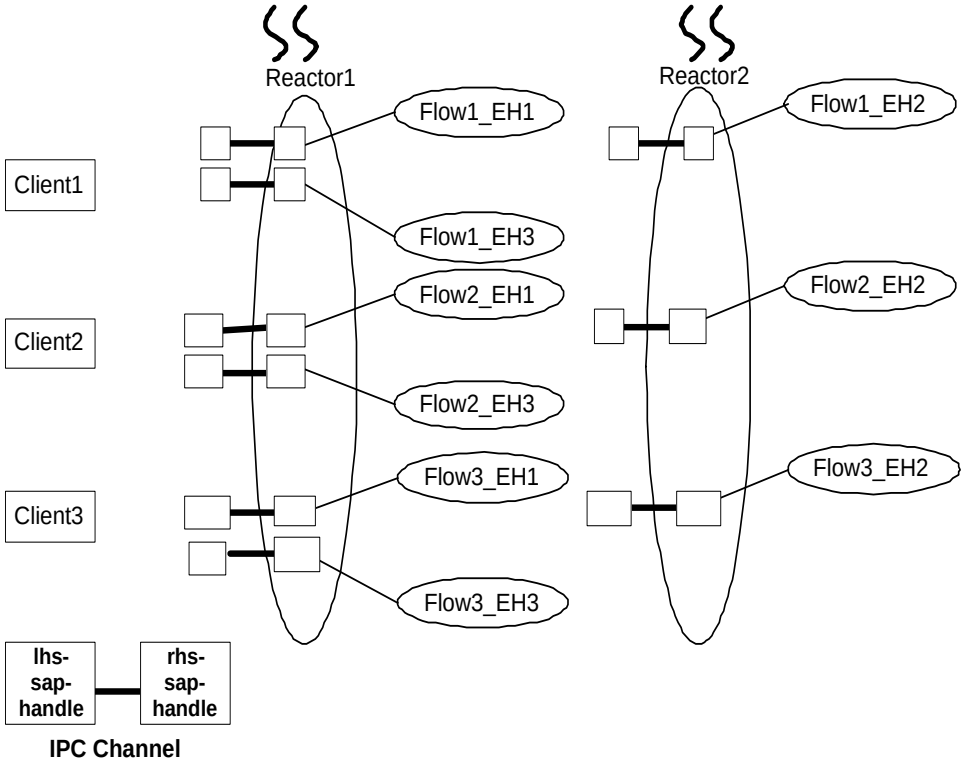


Figure 6.1: Execution Setup for Scenarios

The scenarios that we consider here consist of concurrent call sequences that may span multiple event handlers, each of which is registered with a reactor. There also

may be multiple reactors that host these event handlers. A client originates a *flow*, which is a call-chain spanning possibly multiple event handlers. The communication between clients and event handlers and between event handlers occurs through the IPC channels as is shown in Figure 6.1. This figure shows 3 flows, although the number could vary among scenarios. For the sake of our discussions here, we assume that the same event handler instance is not used by multiple flows, although this is not inherently a restriction of our models. Each event handler has a name that is prefixed with the flow number that uses the event handler. Each client is suffixed with its flow number. Each event handler has an IPC SAP associated with it through which it receives messages sent by a client or another event handler. There are three types of event handlers that we use in our discussions - EH1, EH2 and EH3. EH3 event handlers do not depend on other event handlers for their processing. EH1 and EH2 may depend on other event handlers - based on the scenario, EH1 may depend on EH2 and EH2 may depend on EH3 as part of their service processing. Both EH1 and EH3 type event handlers are hosted in Reactor1, although in some scenarios only one may be present. EH2 type event handlers are hosted in Reactor2. A reactor watches multiple such SAPs for I/O events and then dispatches them to the appropriate event handlers registered with it. The number of threads and the type of reactor (select, thread pool) could vary between scenarios. Different scenarios are realized by setting up clients and event handlers with appropriate SAPs through which communication takes place. Table 6.1 shows the naming convention that we use throughout the discussion of the various scenarios.

Table 6.1: Naming Conventions Used in Discussion of Scenarios

Reactor	Reactor<*reactor num*>
EventHandler	Flow<*flow num*>_EH<*eh num*>
Select Reactor handle_events call stack	Reactor<*reactor num*>_SRHE<*recursion depth*>
TP Reactor handle_events call stack	Reactor<*reactor num*>_TPRHE<*thread num*>
IPC channel	UniDir_IPC_<*lhs SAP handle*>_<*rhs SAP handle*>
Reactor thread	ReactorThread<*thread num*>
Reply Handler	Flow<*flow num*>_EH<*eh num*>_RH
Client	Client<*flow num*>

Since the call stack for handle_events is modeled as a process, it is included in this table. For a select reactor, the label SRHE followed by the stack depth of a nested handle_events call is used and for a thread pool reactor TPRHE is used followed by the number of the reactor thread. Reactor threads are given numbers that are unique across different reactor thread pools.

The following are the key steps during construction of models for the different scenarios: (1) creating the IPC channels, (2) creating the reactors and their threads, (3) creating event handlers, (4) registering event handlers with the appropriate reactor, (5) adding a set of IPC SAPs to the appropriate reactor's I/O watch set, and (6) establishing the connections among clients and event handlers by associating them with the appropriate IPC SAPs.

6.1.1 Modeling the Scenarios in UPPAAL

In Chapter 4, we discussed models of basic building blocks in UPPAAL and how these are constructed using process templates in UPPAAL. Modeling the scenarios involved instantiating these process templates with the appropriate parameters and then establishing communication channels between these automata. All the template processes are instantiated in the "Process Assignments" section of an UPPAAL project. To model the clients in the various scenarios, we used a client process (automaton) template as is shown in Figure 6.2. The client process template takes three parameters - (1) the start time at which the client will send a request to a server, (2) the relative deadline before which the client expects a reply back from the server, and (3) the IPC SAP on which the request is to be sent.

For example, client1 = Client(3, 45, ipc_sap_4) instantiates a client process template which specifies that a request message be sent at time = 3 units and the relative deadline for the reply is 45 time units. The request is to be sent on SAP 4.

Figure 6.2 also shows how we model deadline miss detection by adding two states - Done and DeadlineMiss. Detecting deadline misses thus becomes a reachability query in the model checker. If the client receives the reply within its deadline, then it moves to the Done state, and if not it moves to the DeadlineMiss state. We use the query

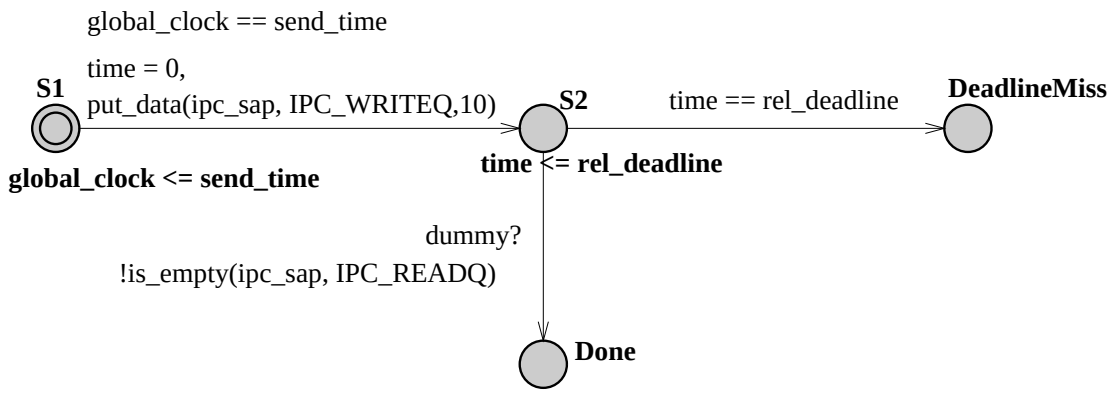


Figure 6.2: Model of Client in UPPAAL

E◇ client1.DeadlineMiss in the UPPAAL verifier to check whether there is a deadline miss. This query asks the following - *Is there any state in any path from the start state, where client1 is in the DeadlineMiss state?*

6.1.2 Modeling the Scenarios in IF

In IF, we created a test harness process that will fork the appropriate model and observer processes for each scenario, and establish associations between clients and SAPs as well as between event handlers and SAPs. The test harness process is supplied with parameters specific to a scenario. For example, some parameters to the test harness are: the type of each of the two reactors, the number of threads in each reactor, the reply wait strategy for event handlers, the SAP on which each client/event handler should send requests, and the SAP on which each event handler should receive requests. This gives us a framework from which to create models for these different scenarios while reusing as much IF code as possible across different scenarios. Since IF does not support the concept of file includes as in C/C++, any IF model has to be self-contained, *i.e.*, it cannot refer to processes in other IF files. This therefore requires duplication of the ACE building block models for each of the different scenarios. To reduce accidental complexities that are inherent in such code duplication, we have developed automated tools (based on GNU make) that combine the common IF models and observers that we discussed in Chapter 5 to compose models for a specific

scenario. From an engineering perspective, this framework enables flexible modeling of multiple scenarios while safely reusing as much code as possible.

The extract in Figure 6.3 shows how we instantiate the test harness to establish a scenario where there is a single client communicating with a single event handler of type EH1, hosted on Reactor1 which is a select reactor with a single thread. The client sends a message to Flow1_EH1 at time = 3 units and its relative deadline for receiving a reply back is at time = 60 units.

```
process Main(1);

var th_params Test_Harness_Params;

state init #start ;
    task th_params.num_flows := 1;
    task th_params.num_ehs := 1;
    task th_params.reactor1_type := RT_SELECT;
    task th_params.reactor1_tp_num := 1;

    task th_params.eh_params[0].more_service := 0;
    task th_params.eh_params[0].reply_wait_strat := NO_WAIT;
    task th_params.eh_params[0].annotation := 0;
    task th_params.eh_params[0].exec_time := 25;

    task th_params.client_params[0].start_time := 3;
    task th_params.client_params[0].rel_deadline := 60;
    task th_params.client_params[0].eh_no_to_connect := 1;

    fork Test_Harness(th_params);
    nextstate done;
endstate;
```

Figure 6.3: IF based Test Harness

The eh_params array is indexed based on the number of the event handler - 0 for EH1 type, 1 for EH2 type and 2 for EH3 type event handlers. The client_params array is indexed by the flow number.

6.2 Execution Traces

One advantage of using executable models is the execution/simulation traces that are produced when we execute our models in a tool like IF or UPPAAL. These traces carry valuable information including, in particular, clues as to what sequence of execution steps led to a specific system state, *e.g.,* a deadlock. This enables us to do more principled analysis of a system rather than just an informal discussion. We now describe the execution trace facilities available in both IF and UPPAAL and how we do various forms of post-processing on these traces to make them suitable for our analysis here.

6.2.1 Execution Traces in UPPAAL

In UPPAAL a property to be verified for the system model is expressed as a temporal logic expression in the UPPAAL verifier. If a property using an existential quantifier (E) is satisfied, then a trace is produced showing the sequence of steps leading to a state where the property is satisfied. There could be multiple execution sequences that could lead to a state where a property is satisfied. On the other hand, if a property using a universal quantifier (A) is not satisfied, then a trace is produced showing the sequence of steps leading to a state where the property is not satisfied.

In UPPAAL, there are two ways to run the verifier: (1) using the GUI, or (2) using the verifyta command line tool that is available with the UPPAAL package. The GUI based trace is useful for replaying execution traces and observing the execution state of the system, but it is cumbersome to use these traces in formal discussions because they mostly depend on visual depiction of information. On the other hand, the verifyta command line tool produces a text-based trace that can be used in formal discussions. We use a combination of these techniques in our discussions here based on their appropriateness to the particular discussion.

The trace file generated by the UPPAAL GUI-based verifier uses its own proprietary format and hence we cannot do any post processing on these traces. These traces are used only by the UPPAAL GUI, for example to replay an execution trace. The trace

file generated by the verifyta tool produces a textual trace, but the format of this is also not well defined and hence not very amenable to post-processing like what we will see for IF. Nevertheless, we use standard Unix utilities like grep and sed to do some post-processing to extract information that is relevant to our discussions here.

We encountered problems collecting model checking statistics in UPPAAL with the verifyta utility in UPPAAL stopping abruptly when doing exhaustive exploration. Hence we were not able to collect statistics for the example scenarios using UPPAAL. However, our development of the models in UPPAAL does show the generality of our modeling architecture.

6.2.2 Execution Traces in IF

The IF explorator provides a facility to produce an execution trace when a property is satisfied or violated. The execution trace contains all execution steps - actions, forks, outputs, function calls, *etc.* We use execution traces to explain formally the behavior that we discuss in the four scenarios. In this section, we discuss in detail the format of these traces and how we post-process the traces produced by IF and make them concise for our needs.

The execution trace produced by IF is in XML format as is shown in Figure 6.4. Each atomic step is surrounded by IFLabel start and end tags. An atomic step is a sequence of steps associated with a single transition. Each atomic step may consist of a sequence of steps each indicated by an XML element of type IFEvent. A task statement in IF, *e.g.,* assigning a value to a variable, appears as an IFEvent XML element with the kind attribute set to IMPORT in the trace. A function call appears as an IFEvent XML element with the kind attribute set to CALL and the value attribute set to the name of the function.

An output statement in IF appears as an IFEvent element with the kind attribute set to OUTPUT and the value attribute set to the IF signal with all of its actual parameters. Trace elements of these kinds give details of the communication messages between IF-processes. Forking a process is logged as an IFEvent element with attribute kind as FORK. The first child pid element of the IFEvent element has the pid of the creator

```
<IfLabel>
<IfEvent kind='IMPORT' value=''>
<by>
<pid name='UniDir_IPC' no='5' /></by>
</IfEvent>
<IfEvent kind='CALL' value='IPC_SAP_enqueue_data'>
<by>
<pid name='UniDir_IPC' no='5' /></by>
</IfEvent>
</IfLabel>
.............
<IfEvent kind='OUTPUT'
value='handle_events{p1={threadid={Thread}3,parent={ThreadPool}2,
caller={ThreadPool}2},p2=3}'>
<from>
<pid name='ThreadPool' no='2' /></from>
<via>
<pid name='Reactor' no='1' /></via>
<to>
<pid name='nil' no='0' /></to>
</IfEvent>
.............
<IfEvent kind='INPUT'
value='handle_events{p1={threadid={Thread}3,parent={ThreadPool}2,
caller={ThreadPool}2},p2=3}'>
<by>
<pid name='Reactor' no='1' /></by>
</IfEvent>
.............
<IfEvent kind='FORK' value='Select_Reactor_Handle_Events'>
<process>
<pid name='Select_Reactor_Handle_Events' no='0' /></process>
<by>
<pid name='Reactor' no='1' /></by>
</IfEvent>
```

Figure 6.4: Trace Output from IF Model Execution

process and the second child pid element has the pid of the created process. We developed post-processing tools to convert the XML based traces into a more readable and concise format for human analysis. For example, the trace that was produced in XML as shown in Figure 6.4 is converted to the format shown in Figure 6.5.

```
41: {ThreadPool}2 ---handle_events(3)---> {Reactor}1
42: {Reactor}1 forks {Select_Reactor_Handle_Events}0
```

Figure 6.5: Trace Output from IF Model Execution After Post-processing

This format is more readable and easier to analyze visually and thus better serves our purpose of illustrating the sequences of events that happen during execution of concurrent processes. For example, the trace in Figure 6.5 shows that the process {ThreadPool}2 sends a message to the process {Reactor}1. A process in IF is denoted by its pid {P}x, where P is the type of the process and x denotes its instance number. The message is of type handle_events and its parameter is 3. Note that here we strip the first parameter of a message from the original XML trace. The first parameter is used to carry context information like the caller, logical thread id, *etc.* This parameter is common to all the IF messages that we use and usually is not relevant to the kind of analysis that we perform here. Nevertheless, our post-processing tools have an option to turn this parameter off or on since this parameter could be useful in some cases, *e.g.,* for debugging our models. The second line in the trace shown in Figure 6.5 shows that a new IF process {Select_Reactor_Handle_Events}0 was forked by a process with pid {Reactor}1.

When many IF processes are spawned, it can be difficult to keep track of the mapping between these processes and the real world entities that they represent. For example, a Select_Reactor_Handle_Events or TP_Reactor_Handle_Events process is spawned by the Reactor process for each handle_events method call. This is done to simulate the thread call stack associated with the method call, as we discussed in Chapters 4 and 5. In our informal discussion of these examples, we use Flow1_EH1, Flow1_EH2, *etc.* to indicate event handlers. All these event handlers are modeled using a single type of IF process called Event_Handler and are instantiated with different parameters to customize their behavior. While analyzing the execution trace, it then becomes difficult to keep track of the mapping between the actual event handlers (Flow1_EH1, Flow1_EH2, *etc.*) and the IF processes ({Event_Handler}0, {Event_Handler}1, *etc.*).

```
{UniDir_IPC}4 : DECLARE_NAME(TYPE_UNIDIR_IPC,{UniDir_IPC}4,8,9)
{Event_Handler}1 : DECLARE_NAME(TYPE_EH,{Event_Handler}1,0,2)
{ThreadPool}0 : DECLARE_NAME(TYPE_TP,{ThreadPool}0,1,0)
{Reactor}0 :DECLARE_NAME(TYPE_TPR_HE,{TP_Reactor_Handle_Events}0,1,1)
```

Figure 6.6: Signal to Map IF Processid to a Name

We address this problem by using a name declaration signal, examples of which are shown in Figure 6.6. A process can declare a name by using the DECLARE_NAME IF signal. This signal has 4 parameters: (1) the type of process; (2) the pid of the process; (3) and (4), one or two process-specific integer parameters. This event is sent to the nil process in IF which does not have any effect other than that this output signal appears in the execution trace. The trace in Figure 6.6 shows some processes declaring their logical names. We now describe the four parameters and the intuition behind each of them. In IF, there is no character string data structure, and hence we have to resort to a different method to associate a name with a process. The type of the process is given as the first parameter, and this helps the post-processing tool to identify the type of the process when it encounters a DECLARE_NAME signal. The type is then used to interpret the next two parameters. The values for the parameters are interpreted as shown in Table 6.2.

Table 6.2: Naming Convention in Post-processed Traces

	Param1(p1)	Param3(p3)	Param4(p4)	Name in trace
Reactor	TYPE_REACTOR	Reactor#	Unused	Reactor<*p3*>
EventHandler	TYPE_EH	Flow#	EH#	Flow<*p3*>_EH<*p4*>
SR_Handle_Events	TYPE_SR_HE	Reactor#	Stack depth	Reactor<*p3*>_SRHE<*p4*>
TPR_Handle_Events	TYPE_TPR_HE	Reactor#	TP Thread#	Reactor<*p3*>_TPRHE<*p4*>
ReplyHandler	TYPE_RH	Flow#	EH#	Flow<*p3*>_EH<*p4*>_RH
ReactorThread	TYPE_TP	Thread#	Unused	ReactorThread<*p3*>
UnidirIPC channel	TYPE_UNIDIR_IPC	lhs-SAP handle	rhs-SAP handle	Unidir_IPC_<*p3*>_<*p4*>
Client	TYPE_CLIENT	Client#	Unused	Client<*p3*>

A # indicates an ordinal number. For example, two reactors in a scenario could be named Reactor1 and Reactor2. A flow is a call chain that starts from a client and possibly may span multiple reactors and event handlers. A ReactorThread is a thread that calls the reactor event loop. A UnidirIPC channel is a unidirectional channel that transfers data from a SAP buffer to another SAP buffer of an IPC channel. For the SR_Handle_Events process, the fourth parameter is the depth of recursion of the handle_events call. This is useful for analyzing traces in the case of waiting for a reply using the WaitOnReactor strategy.

After mapping the IF pids to names, the trace becomes more readable for analysis. For example, part of a trace is shown in Figure 6.7 that uses IF pids to show the interactions between processes and the sequence of these interactions. The same trace after mapping IF pids to names is shown in Figure 6.8.

```
{TP_Reactor_Handle_Events}0 ---handle_input(2)--->{Event_Handler}0
{Select_Reactor_Handle_Events}0 ---handle_input(4)--->{Event_Handler}1
{TP_Reactor_Handle_Events}1 ---handle_input(6)--->{Event_Handler}2
{Event_Handler}2 ---handle_input_return(0)--->{TP_Reactor_Handle_Events}1
```

Figure 6.7: IF Trace Before Pid to Name Mapping

```
1: Reactor1_TPRHE1 ---handle_input(2)---> Flow1_EH1
2: Reactor2_SRHE1 ---handle_input(4)---> Flow1_EH2
3: Reactor1_TPRHE2 ---handle_input(6)---> Flow1_EH3
4: Flow1_EH3 ---handle_input_return(0)---> Reactor1_TPRHE2
```

Figure 6.8: IF Trace After Pid to Name Mapping

Note that during post processing this capability can be turned on or off so that the trace without names can still be obtained if needed, *e.g.,* for debugging our models. Line 1 in the trace in Figure 6.8 shows that the first thread in the thread pool of Reactor1 - a TP reactor - calls the handle_input method on the Flow1_EH1 event handler. Line 2 shows that Reactor2 - a select reactor - calls the handle_input method on the Flow1_EH2 event handler. Line 3 shows that the second thread in the thread pool of Reactor1 calls handle_input on the Flow1_EH3 event handler. Line 4 shows that the Flow1_EH3 event handler returns from the handle_input call and the flow of control goes back to the second thread in the thread pool of Reactor1.

6.3 Scenarios

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 how interference between different flows' execution can arise for different middleware policy and mechanism choices, and to show how in each case the particular form of interference can be analyzed through model checking. Note that these scenarios capture a small but representative set of design choices that are available when configuring the middleware infrastructure for an application. For example, possible design solutions

for communication in these scenarios include two-way calls, one-way calls with different messaging options [75] (SYNC_WITH_TARGET, SYNC_WITH_SERVER, *etc.*), Asynchronous Method Invocation [98], and Asynchronous Message Handling [24]. Our models capture the semantics of common middleware building blocks with which these different design solutions can be realized, thus giving our approach broad applicability.

6.4 Scenario 1 - Blocking in a Single Reactor

In real-time and embedded systems, crucial system properties can involve timing constraints such as receiving the result of a method invocation before a relative deadline. In this scenario, we consider a case where system timing is affected by interference between nominally independent call sequences, when they must contend for resources such as use of a single reactor thread. Figure 6.9 shows the setup for this scenario in which there are two flows and in each flow a client sends a message to an event handler of type EH1 hosted on Reactor1 and this event handler sends a reply back to the client after doing some processing.

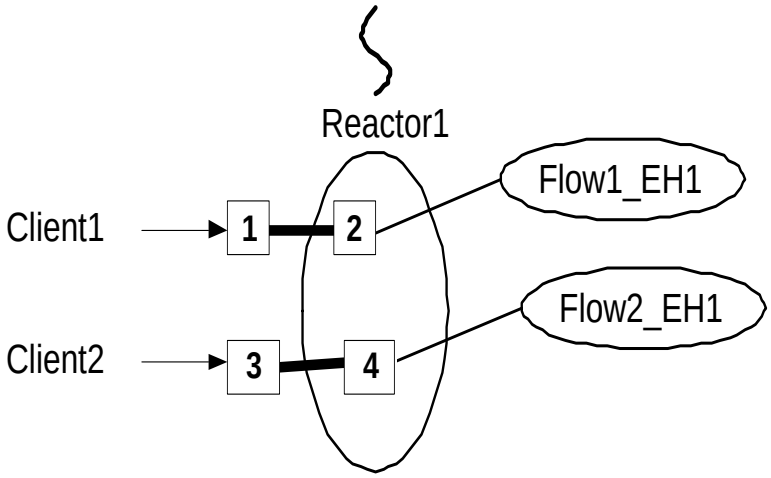


Figure 6.9: Scenario 1 Setup

Figure 6.10 shows two call sequences in which Client1 and Client2 invoke methods and receive replies from event handlers Flow1_EH1 and Flow2_EH1 respectively. This figure shows the timelines for each of the individual call sequences - Client1→Flow1_EH1 and Client2→Flow2_EH1 - in isolation without considering the interleaving of calls.

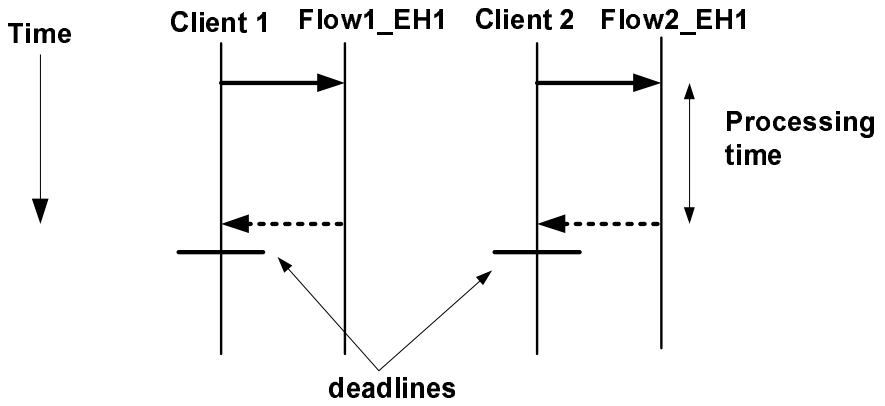


Figure 6.10: Call Sequence for Scenario 1

Informal Analysis. In this scenario a single thread is used by a reactor to demultiplex events to its registered event handlers. Since both Flow1_EH1 and Flow2_EH1 are deployed on the same single-threaded reactor, as shown in Figure 6.9, they can only handle events sequentially. If the request messages from Client1 to Flow1_EH1 and Client2 to Flow2_EH1 arrive at the server at roughly the same time, then whichever event handler is dispatched first will delay the other event handler, potentially resulting in a missed deadline. 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. In this scenario, blocking delays are caused by the implicit serialization caused by the single threaded reactor. Later on in scenario 3, we will see another kind of blocking factor that is caused by the nesting of upcalls from the same reactor. Now we state the following hypothesis based on the informal analysis above.

Hypothesis 1. *Multiple computation flows passing through the same single threaded reactor could interfere with each other and cause blocking delays that could result in missed deadlines.*

We now show how we can analyze this scenario using our executable formal models in both UPPAAL and IF to evaluate the above hypothesis.

6.4.1 Formal Analysis of Scenario 1 in UPPAAL

We composed the model for Scenario 1 in UPPAAL using the models of building blocks that we described in Chapter 4. The UPPAAL process templates are instantiated as in Figure 6.11.

```
Unidir_IPC_1_2 = IPC_Channel(ipc_sap_1,ipc_sap_2,0);
Unidir_IPC_2_1 = IPC_Channel(ipc_sap_2,ipc_sap_1,0);
Unidir_IPC_3_4 = IPC_Channel(ipc_sap_3,ipc_sap_4,0);
Unidir_IPC_4_3 = IPC_Channel(ipc_sap_4,ipc_sap_3,0);

Flow1_EH1 = EventHandler(FLOW1_EH1, 25, ipc_sap_2, REACTOR1);
Flow2_EH1 = EventHandler(FLOW2_EH1, 25, ipc_sap_4, REACTOR1);

Reactor1_SRHE0 = Select_Reactor(REACTOR1,
                          handle_events_channels[REACTOR1],
                          handle_events_return_channels[REACTOR1],
                          reactor_states[REACTOR1]);

ReactorThread1 = ReactorThread(handle_events_channels[REACTOR1],
                          handle_events_return_channels[REACTOR1]);

Client1 = Client(3, 45, ipc_sap_1);
Client2 = Client(3, 45, ipc_sap_3);
```

Figure 6.11: Instantiating Scenario 1 in UPPAAL

Two channels are created to represent the connections Client1→Flow1_EH1 and Client2→Flow2_EH1. Both Flow1_EH1 and Flow2_EH1 are created with computation times of 25 units. Each of them is registered with Reactor1 which is the only reactor in this scenario. Flow1_EH1 handles events occurring in ipc_sap_2 and Flow2_EH1 handles events occurring in ipc_sap_4. We use a select reactor in this scenario to illustrate the blocking factors caused by the implicit serialization of event dispatching that occurs in this reactor. Finally, two clients are instantiated. Each

client sends a request message 3 time units from the start of model execution. We have modeled the clients so that each of them sends a request message to their respective event handlers exactly when time = 3 after system initialization, thereby achieving coordination between the client and server. Each client has a deadline of 45 time units relative to the sending of the request message before which it expects a reply message. Client1 and Client2 send their request messages on channels ipc_sap_1 and ipc_sap_3 respectively.

The temporal logic expression E◇ Client1.DeadlineMiss or Client2.DeadlineMiss is used to verify whether there are any deadline misses. This query asks - *Is there any state in any path from the start state, where Client1 is in the DeadlineMiss state or Client2 is in the DeadlineMiss state?* With the composed model shown above, the verifier shows that this property is satisfied. Note that the verifier stops further state space exploration as soon as it finds a state that satisfies this property. Figure 6.12 shows a textual trace (using the verifyta tool) of the sequence of events that led to the above property being satisfied.

The system model starts execution by calling the init_func function (line 1) that registers the event handlers with the appropriate reactors and adds the SAPs to the appropriate reactor watch set. The reactor thread calls the handle_events method (lines 2-3) which starts watching SAP handles 2 and 4 for read events. After time progresses by 3 units (line 4), Client2 sends a message (line 5) through SAP handle 3. Note that the trace does not show the actual parameters. For example, the put_data function in line 5 does not show the actual parameter ipc_sap_3 with which the Client2 automaton is instantiated. In contrast, the GUI simulator shows the automata with the actual parameters with which they are instantiated. The unidirectional IPC channel from SAP handle 3 to 4 transfers data (lines 6,7) from the write SAP buffer for 3 to the read SAP buffer for 4. Client1 then sends a message and the same sequence as above is repeated (lines 8-10). Reactor1 then unblocks (line 11) from its wait for I/O events, since two SAP handles are ready for reading. The reactor picks the first SAP handle (line 12) and makes an upcall to the appropriate event handler (line 13). After receiving the upcall (line 14), the event handler Flow1_EH1 does some CPU bound computation for 25 time units (line 15) and then writes a reply message (line 16) to the write buffer for SAP handle 2. The message is transferred to read buffer for SAP handle 1 by a unidirectional IPC channel (lines 17,20). The

```
1   InitProcess.S1->InitProcess.S2 { 1, tau, init_func() }
2   ReactorThread1.S1->ReactorThread1.S2 { 1, reactor_handle_events!, 1 }
3   Reactor1_SRHE0.S1->Reactor1_SRHE0.S2 { 1, handle_events?, 1 }
4 Delay: 3
5   Client2.S1->Client2.S2 { global_clock == send_time, tau, time := 0, p
5 ut_data(ipc_sap, IPC_WRITEQ, 10) }
6   Unidir_IPC_3_4.S1->Unidir_IPC_3_4.S2 { !is_empty(lhs_sap, IPC_WRITEQ)
6 , dummy?, t := 0 }
7   Unidir_IPC_3_4.S2->Unidir_IPC_3_4.S1 { t == prop_delay, tau, bytes_re
7 ad := get_data(lhs_sap, IPC_WRITEQ), put_data(rhs_sap, IPC_READQ, bytes
7 _read) }
8   Client1.S1->Client1.S2 { global_clock == send_time, tau, time := 0, p
8 ut_data(ipc_sap, IPC_WRITEQ, 10) }
9   Unidir_IPC_1_2.S1->Unidir_IPC_1_2.S2 { !is_empty(lhs_sap, IPC_WRITEQ)
9 , dummy?, t := 0 }
10  Unidir_IPC_1_2.S2->Unidir_IPC_1_2.S1 { t == prop_delay, tau, bytes_re
10 ad := get_data(lhs_sap, IPC_WRITEQ), put_data(rhs_sap, IPC_READQ, bytes
10 _read) }
11  Reactor1_SRHE0.S2->Reactor1_SRHE0.S3 { is_any_sap_hot(reactor_state.r
11 ead_sap_set_to_watch, reactor_state.write_sap_set_to_watch), dummy?, ho
11 t_read_sap_set := get_hot_read_saps(reactor_state.read_sap_set_to_watch
11 ), hot_write_sap_set := get_hot_write_saps(reactor_state.write_sap_set_
11 to_watch) }
12  Reactor1_SRHE0.S3->Reactor1_SRHE0.S4 { size(hot_read_sap_set) > 0, ta
12 u, pop_first_sap(hot_read_sap_set, first_hot_sap), upcall_handler := ge
12 t_handler(reactor_state.handler_repo, first_hot_sap) }
13  Reactor1_SRHE0.S4->Reactor1_SRHE0.S5 { 1, eh_hi_channels[upcall_handl
13 er]!, 1 }
14  Flow1_EH1.S1->Flow1_EH1.S2 { 1, eh_hi_channels[eh_pid]?, t := 0, get_
14 data(ipc_sap, IPC_READQ) }
15 Delay: 25
16  Flow1_EH1.S2->Flow1_EH1.S3 { t == comp_time, tau, put_data(ipc_sap, I
16 PC_WRITEQ, 10) }
17  Unidir_IPC_2_1.S1->Unidir_IPC_2_1.S2 { !is_empty(lhs_sap, IPC_WRITEQ)
17 , dummy?, t := 0 }
18  Flow1_EH1.S3->Flow1_EH1.S1 { 1, eh_hir_channels[eh_pid]!, 1 }
19  Reactor1_SRHE0.S5->Reactor1_SRHE0.S3 { 1, eh_hir_channels[upcall_hand
19 ler]?, 1 }
20  Unidir_IPC_2_1.S2->Unidir_IPC_2_1.S1 { t == prop_delay, tau, bytes_re
20 ad := get_data(lhs_sap, IPC_WRITEQ), put_data(rhs_sap, IPC_READQ, bytes
20 _read) }
21  Client1.S2->Client1.Done { !is_empty(ipc_sap, IPC_READQ), dummy?, 1 }
22  Reactor1_SRHE0.S3->Reactor1_SRHE0.S4 { size(hot_read_sap_set) > 0, ta
22 u, pop_first_sap(hot_read_sap_set, first_hot_sap), upcall_handler := ge
22 t_handler(reactor_state.handler_repo, first_hot_sap) }
23  Reactor1_SRHE0.S4->Reactor1_SRHE0.S5 { 1, eh_hi_channels[upcall_handl
23 er]!, 1 }
24  Flow2_EH1.S1->Flow2_EH1.S2 { 1, eh_hi_channels[eh_pid]?, t := 0, get_
24 data(ipc_sap, IPC_READQ) }
25 Delay: 20
26  Client2.S2->Client2.DeadlineMiss { time == rel_deadline, tau, 1 }
```

Figure 6.12: Scenario 1 Trace in UPPAAL

handle_input upcall returns (lines 18-19). Client1 receives the reply (line 21) and is done. Reactor1 then iterates to the next ready handle (line 22) and makes an upcall (line 23) to Flow2_EH1 which in turn receives the upcall (line 24) and does some processing (line 25) for 20 time units. At this time the deadline for Client2 expires causing it to take a transition to the DeadlineMiss state.

In contrast to the previous query, the query - E◇ client1.Done and client2.Done - forces the UPPAAL verifier to do an exhaustive state space exploration since the verifier finds that this property is not satisfied. This query means - *Is there any state in any path from the start state, where Client1 is in the Done state and Client2 is in the Done state?*. The exhaustive exploration output from the verifyta tool is shown in Figure 6.13.

```
$ verifyta -N -d -t 1 single_reactor_bf.xml single_reactor_bf.q2
UPPAAL version 3.6 Alpha 3-pre1, Dec 2005 -- verifyta.
Compiled with g++-4.0.2 -DNDEBUG -O2 -march=pentiumpro -Wall.
Copyright (c) 1995 - 2005, Uppsala University and Aalborg University.
All rights reserved.
Options for the verification:
  Diagnostic trace is on
  Search order is depth first
  Using conservative space optimisation
  State space representation uses minimal constraint systems

Verifying property 1 at line 1
 -- Property is NOT satisfied.
```

Figure 6.13: Scenario 1 Exhaustive Exploration in UPPAAL

6.4.2 Formal Analysis of Scenario 1 in IF

The trace in Figure 6.14 shows the sequence of steps leading to a missed deadline for one of the clients after the expiry of its deadline of 45 time units relative to its sending of request to an event handler registered with reactor1. The complete trace is longer than this and a lot of the initial steps are related to construction of model elements and their relationships. For example, creating the reactors, creating the IPC channels, creating the event handlers, registering the event handlers with the appropriate reactors are all part of the initial system initialization. We don't show this here for the sake of clarity and brevity. Instead we have already shown the static

structure of the system in Figure 6.9, which will provide the context for our analysis of the trace for this scenario.

```
1: {Test_Harness}0 ---INIT_MODE_DONE()---> {nil}0
2: Time advanced by 3 units. Global time is 3
3: Client1 : TRACE_SAP_Buffer_Write(1,10)
4: Client2 : TRACE_SAP_Buffer_Write(3,10)
5: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
6: Unidir_IPC_3_4 : TRACE_SAP_Buffer_Transfer(3,4,10)
7: Reactor1_SRHE0 ---handle_input(2)---> Flow1_EH1
8: Time advanced by 25 units. Global time is 28
9: Flow1_EH1 : TRACE_SAP_Buffer_Write(2,10)
10: Flow1_EH1 ---handle_input_return(0)---> Reactor1_SRHE0
11: Unidir_IPC_2_1 : TRACE_SAP_Buffer_Transfer(2,1,10)
12: Client1 : TRACE_SAP_Buffer_Read(1,10)
13: Reactor1_SRHE0 ---handle_input(4)---> Flow2_EH1
14: Time advanced by 21 units. Global time is 49
15: Client2 : TRACE_DeadlineMiss()
{u'{UniDir_IPC}1': u'Unidir_IPC_2_1', u'{UniDir_IPC}0':
u'Unidir_IPC_1_2', u'{UniDir_IPC}3': u'Unidir_IPC_4_3',
u'{UniDir_IPC}2': u'Unidir_IPC_3_4', u'{Event_Handler}1':
u'Flow2_EH1', u'{Event_Handler}0': u'Flow1_EH1', u'{Reactor}0':
u'Reactor1', u'{Select_Reactor_Handle_Events}0': u'Reactor1_SRHE0',
u'{ThreadPool}1': u'ReactorThread2', u'{ThreadPool}0':
u'ReactorThread1', u'{ClientProc}0': u'Client1', u'{ClientProc}1':
u'Client2'}
```

Figure 6.14: Scenario 1 Trace in IF

Line 1 shows that the system initialization mode is done. At time = 3 units (line 2), Client1 writes a request message (line 3) to the SAP with handle 1 and Client2 writes a request message (line 4) to the SAP with handle 3. In this example, the lhs-SAP handle for the IPC channel from Client1 to Flow1_EH1 is 1 and the rhs-SAP handle for the same channel is 2. In this case, we refer to the forward unidirectional channel as 1-2 and the reverse unidirectional channel as 2-1. The channel 1-2 is used by Client1 to send a message to Flow1_EH1 and channel 2-1 is used by Flow1_EH1 to send a message back to Client1. The data transfer is done by the two UniDir_IPC channel automata that are associated with each IPC channel. Lines 5 and 6 show that the forward channel automata associated with the two IPC channels (1-2 and 3-4) transfer data from handles 1 to 2 and 3 to 4 respectively. Reactor1 detects the I/O events on handles 2 and 4 and proceeds to make the upcall as shown in Line 7. Note that even though we say that the reactor makes an upcall, the upcall itself is done in the context of the handle_events method which in the model is represented

as an IF process and hence this appears as Reactor1_SRHE0 in the trace. The first upcall is made to Flow1_EH1 that is registered with the reactor to handle events on handle 2. Flow1_EH1 does some computation that takes 25 time units and we see time advancing by 25 time units in Line 8. Once the computation is done, Flow1_EH1 writes a reply on to handle 2 and returns control to the reactor in Line 10. In Line 11, the data is transferred from handle 2 to 1 and the Client1 reads the reply from handle 1 in Line 12. The reactor now proceeds to make another upcall corresponding to the read-ready handle 4 (Line 13), which tries to perform a computation for 25 time units. When time has advanced by 21 time units (Line 14) Client2 misses its deadline (Line 15) and this causes Client2 to enter an error state which is monitored by an observer that in turn stops further state exploration. In this example, each client process has a relative deadline of 45 time units. Taking into account the start time of 3 time units, at 48 time units if a client does not already get a reply back from its corresponding event handler, then a deadline miss occurs. The time from the start of the execution is indicated by the global time, as is shown in Line 14. After the end of the trace, the pid-to-name mapping that is stored internally during post-processing is shown. This is useful mostly for debugging purposes only and hence we will omit this part for the subsequent traces.

We increased the relative deadline of the clients to 60 and observed the trace output from execution of our model shown in Figure 6.15 which shows that there is no deadline miss.

```
13: Reactor1_SRHE0 ---handle_input(4)---> Flow2_EH1
14: Time advanced by 25 units. Global time is 53
15: Flow2_EH1 : TRACE_SAP_Buffer_Write(4,10)
16: Flow2_EH1 ---handle_input_return(0)---> Reactor1_SRHE0
17: Reactor1_SRHE0 ---handle_events_return()---> ReactorThread1
18: ReactorThread1 ---handle_events(1)---> Reactor1
19: Reactor1 forks {Select_Reactor_Handle_Events}1
20: Unidir_IPC_4_3 : TRACE_SAP_Buffer_Transfer(4,3,10)
21: Client2 : TRACE_SAP_Buffer_Read(3,10)
```

Figure 6.15: Scenario 1 Trace in IF with Later Deadline

This trace is the same as in Figure 6.14 until Line 13. After the upcall to Flow2_EH, time advances by 25 units and Flow2_EH1 completes its computation and writes its reply to handle 4 (line 15) and returns control to the reactor (line 16). Now the flow of control returns to the reactor thread (line 17) which had originally called

the handle_events method on Reactor1. The reactor thread (ReactorThread1) continues in a loop calling handle_events again on Reactor1. Note that a new Select_Reactor_Handle_Events process is forked to represent the new handle_events call stack. The reply is transferred from handle 4 to handle 3 (line 20), which is then read by Client2 (line 21).

To illustrate that the IF model checker explores different interleavings we show another trace in Figure 6.16 that leads to no deadline misses.

```
13: Reactor1_SRHE0 ---handle_events_return()---> ReactorThread1
14: ReactorThread1 ---handle_events(1)---> Reactor1
15: Reactor1 forks {Select_Reactor_Handle_Events}1
16: Reactor1_SRHE0 ---handle_input(4)---> Flow2_EH1
17: Time advanced by 25 units. Global time is 53
18: Flow2_EH1 : TRACE_SAP_Buffer_Write(4,10)
19: Flow2_EH1 ---handle_input_return(0)---> Reactor1_SRHE0
20: Reactor1_SRHE0 ---handle_events_return()---> ReactorThread1
21: Unidir_IPC_4_3 : TRACE_SAP_Buffer_Transfer(4,3,10)
22: Client2 : TRACE_SAP_Buffer_Read(3,10)
```

Figure 6.16: A Different Scenario 1 Trace in IF with Later Deadline

In Figure 6.15, after making the first upcall the reactor makes the second upcall without returning control back to the reactor thread. Only after the second upcall is completed, the flow of control returns back to the reactor thread (line 17 in Figure 6.15). In Figure 6.16, the flow of control returns to the reactor thread (line 13) after the first upcall is done. The handle_events method is invoked again and an upcall is made for the read-ready handle 4 (lines 14-16). The sequence of execution shown after this is similar to that in Figure 6.15. The change in this execution sequence is potentially due to the different interleavings that are possible among the following execution steps - (1) the message from handle 1 written to handle 2, (2) the message from handle 3 written to handle 4, and (3) the reactor coming out of its wait on multiple I/O events. If (3) happens after (1) and (2) then the sequence of execution looks like Figure 6.15, whereas if (3) happens between (1) and (2), the the execution looks like Figure 6.16. In order to prove that this is indeed the case, we added a new log event (TRACE_Reactor_IO_Wait_Done) to our model, which occurs when the reactor unblocks from its wait on multiple I/O handles. As part of this log event we record the read-ready and write-ready handles, to give us a clear indication that our hypothesis above is indeed valid. This process also illustrates a methodology for how

to use and extend our models to support further hypotheses. Extracts from the new traces after the addition of the new event are shown in Figure 6.17.

```
Interleaving 1:
5: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
6: Unidir_IPC_3_4 : TRACE_SAP_Buffer_Transfer(3,4,10)
7: Reactor1_SRHE0 : TRACE_Reactor_IO_Wait_Done({2,4,},{})

Interleaving 2:
5: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
6: Reactor1_SRHE0 : TRACE_Reactor_IO_Wait_Done({2,},{})
....(upcall to Flow1_EH1 and return control to reactor thread)
15: ReactorThread1 ---handle_events(1)---> Reactor1
16: Reactor1 forks {Select_Reactor_Handle_Events}1
17: Reactor1_SRHE0 : TRACE_Reactor_IO_Wait_Done({4,},{})
```

Figure 6.17: Scenario 1 Traces with New Log Event Added

In Interleaving 1 in Figure 6.17, both handles 3 and 4 are read-ready when the reactor returns from its wait (line 7), whereas in Interleaving 2, the reactor returns from its wait (line 6) as soon as handle 2 becomes read-ready. Only after the upcall corresponding to handle 2 was completed and the next iteration of handle_events started (line 16) did the reactor detect the readiness of handle 4 (line 17). The traces in Figure 6.17 thus show that the model checker explores different execution sequences.

6.5 Scenario 2 - Multiple Reactors, *WaitOnConnection* strategy

In addition to analyzing interference arising from direct contention between handlers for a single resource, it is important to evaluate more complex interference scenarios involving sequences of inter-dependent actions. In this example, we show how timing properties of the system are affected not only by interfering call sequences, but also by the strategy used to wait for replies from remote function calls. Consider for example another scenario based on the setup shown in Figure 6.18 in which a call chain spans all three handlers, with Flow1_EH1 depending on Flow1_EH2 and Flow1_EH2 depending on Flow1_EH3.

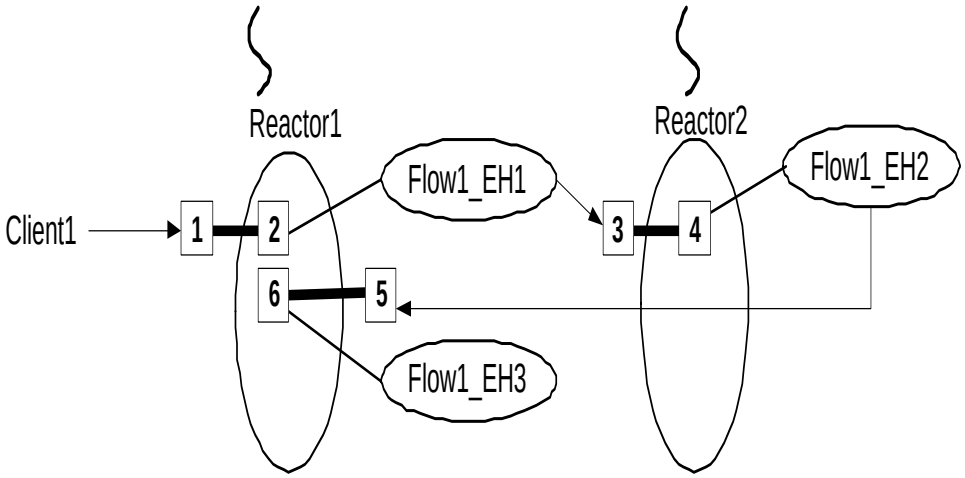


Figure 6.18: Setup for Scenario 2

Informal Analysis. Flow1_EH1 and Flow1_EH3 are registered with the same reactor and this reactor has a single thread. With the WaitOnConnection reply wait strategy, the single thread is already in an upcall (to Flow1_EH1) when there is another incoming request for Flow1_EH3. This is called a *nested upcall*. Because of 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 Flow1_EH2 to Flow1_EH3. Figure 6.19 shows the relevant call sequence. The deadline for Client1 is missed when no progress can be made by the system after the request is sent from Flow1_EH2 to Flow1_EH3. We now state the following hypothesis based on this informal analysis.

Hypothesis 2.1. *Nested upcalls will cause a deadlock in the context of a single-threaded select reactor and WaitOnConnection reply wait strategy.*

A possible solution to eliminate the deadlock is to use a thread pool reactor and increase the number of threads in the reactor so that looping calls can be handled without deadlocking the system. The thread pool reactor uses the Leader/Followers pattern and even if one thread from the thread pool (a follower) is busy making an upcall to one of the event handlers, another thread (the leader) can be waiting for I/O events. This leads to the following new hypothesis.

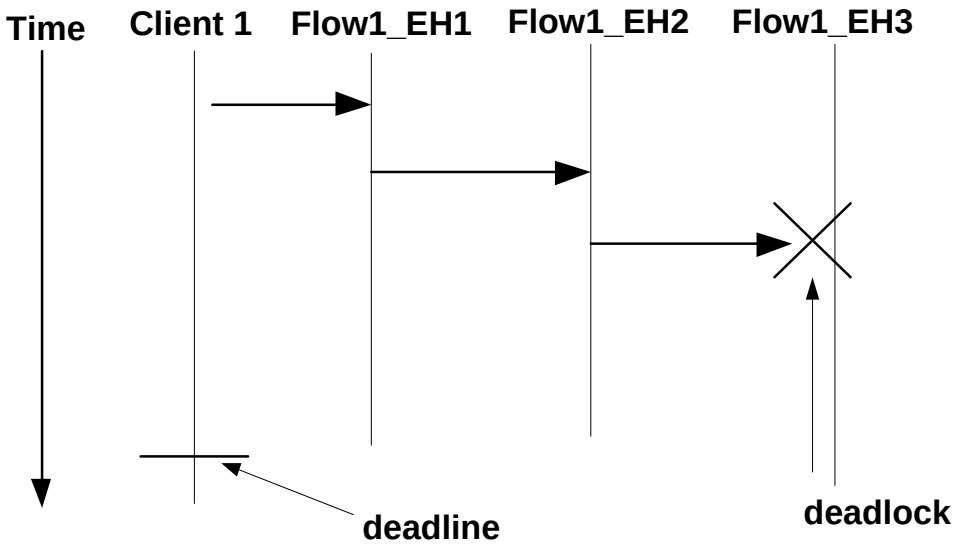


Figure 6.19: Call sequence for Scenario 2

Hypothesis 2.2. *Deadlocks with nested upcalls in the context of WaitOnConnection reply wait strategy can be eliminated by using a thread pool reactor and increasing the number of threads in the reactor to $k+1$, where k is the number of cycles in the call graph, assuming there are no concurrent calls to the root of the call graph.*

6.5.1 Formal Analysis of Scenario 2 in UPPAAL

For Scenario 2, the UPPAAL model templates are instantiated as in Figure 6.20. In this scenario, there are 3 bidirectional channels - Client1 to Flow1_EH1, Flow1_EH1 to Flow1_EH2 and Flow1_EH2 to Flow1_EH3. Three event handlers are instantiated, each with a computation time of 25 time units. Flow1_EH1 and Flow2_EH2 take two SAPs as parameters - the first SAP (third template parameter) is the SAP on which the event handler expects its request message; the second SAP (fourth template parameter) is the SAP with which the event handler sends further requests to another event handler as part of the call sequence. For example, Flow1_EH1 receives request messages on ipc_sap_2 and sends request messages to Flow1_EH2 on ipc_sap_3. The IPC_Channel Unidir_IPC_3_4 forwards this message from ipc_sap_3 to ipc_sap_4, which in turn is used by Flow1_EH2 to receive its request messages. Two reactors are instantiated with their respective UPPAAL channels for communication

```
Flow1_EH1 = EventHandler1(FLOW1_EH1, 25, ipc_sap_2, ipc_sap_3, REACTOR1);
Flow1_EH2 = EventHandler2(FLOW1_EH2, 25, ipc_sap_4, ipc_sap_5, REACTOR2);
Flow1_EH3 = EventHandler3(FLOW1_EH3, 25, ipc_sap_6, REACTOR1);

Reactor1_SRHE0 = Select_Reactor(REACTOR1,
             handle_events_channels[REACTOR1],
             handle_events_return_channels[REACTOR1],
             reactor_states[REACTOR1]);

Reactor2_SRHE0 = Select_Reactor(REACTOR2,
                     handle_events_channels[REACTOR2],
                     handle_events_return_channels[REACTOR2],
                     reactor_states[REACTOR2]);

ReactorThread1 = ReactorThread(handle_events_channels[REACTOR1],
                           handle_events_return_channels[REACTOR1]);

ReactorThread2 = ReactorThread(handle_events_channels[REACTOR2],
                           handle_events_return_channels[REACTOR2]);

Client1 = Client(3, 1000, ipc_sap_1);
```

Figure 6.20: Scenario 2 Instantiation in UPPAAL

with other automata. Each reactor is instantiated with its corresponding state. Two thread automata are instantiated, each of which initiates the event loop by calling handle_events on each reactor. Finally the Client automaton is instantiated with a relative deadline of 1000 time units. Note that a disproportionately high value for the relative deadline is chosen so that a deadline miss gives further evidence of a deadlock and can be used to produce a trace leading to the deadline miss.

We used the query E◇ Client1.Done in the UPPAAL verifier to verify whether Client1 ever gets a reply back from Flow1_EH1. This query causes the verifier to do an exhaustive search of the state space since it finds that the property is not satisfied as shown in Figure 6.21.

```
$ verifyta -N -d -t 1 woc_deadlock.xml woc_deadlock.q1

Verifying property 1 at line 1
 -- Property is NOT satisfied.
```

Figure 6.21: UPPAAL verifyta output for Scenario 2 with 1 Thread

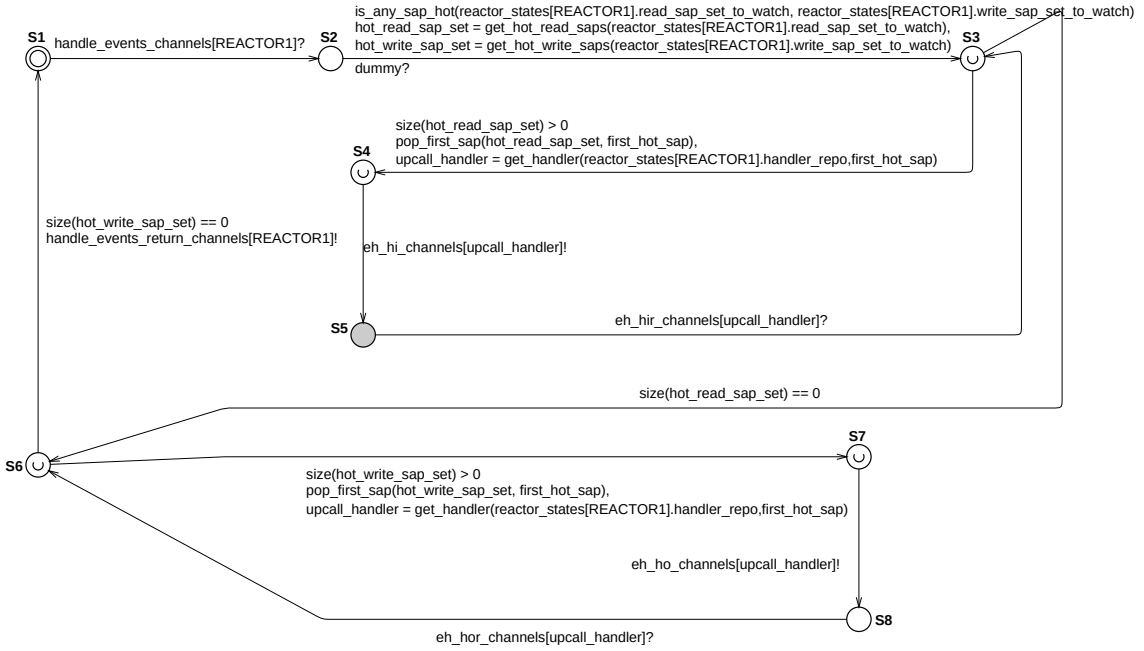
```
 7   Client1.S1->Client1.S2 { global_clock == send_time, tau, time := 0, p
 7 ut_data(ipc_sap, IPC_WRITEQ, 10) }
 8   Unidir_IPC_1_2.S1->Unidir_IPC_1_2.S2 { !is_empty(lhs_sap, IPC_WRITEQ)
 8 , dummy?, t := 0 }
 9   Unidir_IPC_1_2.S2->Unidir_IPC_1_2.S1 { t == prop_delay, tau, bytes_re
 9 ad := get_data(lhs_sap, IPC_WRITEQ), put_data(rhs_sap, IPC_READQ, bytes
 9 _read) }
10   Reactor1_SRHE0.S2->Reactor1_SRHE0.S3 { is_any_sap_hot(reactor_state.r
10 ead_sap_set_to_watch, reactor_state.write_sap_set_to_watch), dummy?, ho
10 t_read_sap_set := get_hot_read_saps(reactor_state.read_sap_set_to_watch
10 ), hot_write_sap_set := get_hot_write_saps(reactor_state.write_sap_set_
10 to_watch) }
11   Reactor1_SRHE0.S3->Reactor1_SRHE0.S4 { size(hot_read_sap_set) > 0, ta
11 u, pop_first_sap(hot_read_sap_set, first_hot_sap), upcall_handler := ge
11 t_handler(reactor_state.handler_repo, first_hot_sap) }
12   Reactor1_SRHE0.S4->Reactor1_SRHE0.S5 { 1, eh_hi_channels[upcall_handl
12 er]!, 1 }
13   Flow1_EH1.S1->Flow1_EH1.S2 { 1, eh_hi_channels[eh_pid]?, t := 0, get_
13 data(in_ipc_sap, IPC_READQ) }
14 Delay: 25
15   Flow1_EH1.S2->Flow1_EH1.S3 { t == comp_time, tau, put_data(out_ipc_sa
15 p, IPC_WRITEQ, 10) }
16   Unidir_IPC_3_4.S1->Unidir_IPC_3_4.S2 { !is_empty(lhs_sap, IPC_WRITEQ)
16 , dummy?, t := 0 }
17   Unidir_IPC_3_4.S2->Unidir_IPC_3_4.S1 { t == prop_delay, tau, bytes_re
17 ad := get_data(lhs_sap, IPC_WRITEQ), put_data(rhs_sap, IPC_READQ, bytes
17 _read) }
18   Reactor2_SRHE0.S2->Reactor2_SRHE0.S3 { is_any_sap_hot(reactor_state.r
18 ead_sap_set_to_watch, reactor_state.write_sap_set_to_watch), dummy?, ho
18 t_read_sap_set := get_hot_read_saps(reactor_state.read_sap_set_to_watch
18 ), hot_write_sap_set := get_hot_write_saps(reactor_state.write_sap_set_
18 to_watch) }
19   Reactor2_SRHE0.S3->Reactor2_SRHE0.S4 { size(hot_read_sap_set) > 0, ta
19 u, pop_first_sap(hot_read_sap_set, first_hot_sap), upcall_handler := ge
19 t_handler(reactor_state.handler_repo, first_hot_sap) }
20   Reactor2_SRHE0.S4->Reactor2_SRHE0.S5 { 1, eh_hi_channels[upcall_handl
20 er]!, 1 }
21   Flow1_EH2.S1->Flow1_EH2.S2 { 1, eh_hi_channels[eh_pid]?, t := 0, get_
21 data(in_ipc_sap, IPC_READQ) }
22 Delay: 25
23   Flow1_EH2.S2->Flow1_EH2.S3 { t >= comp_time, tau, t := 0, put_data(ou
23 t_ipc_sap, IPC_WRITEQ, 10) }
24   Unidir_IPC_5_6.S1->Unidir_IPC_5_6.S2 { !is_empty(lhs_sap, IPC_WRITEQ)
24 , dummy?, t := 0 }
25   Unidir_IPC_5_6.S2->Unidir_IPC_5_6.S1 { t == prop_delay, tau, bytes_re
25 ad := get_data(lhs_sap, IPC_WRITEQ), put_data(rhs_sap, IPC_READQ, bytes
25 _read) }
26 Delay: 950
27   Client1.S2->Client1.DeadlineMiss { time == rel_deadline, tau, 1 }
```

Figure 6.22: UPPAAL Trace Showing Sequence Leading to Deadlock

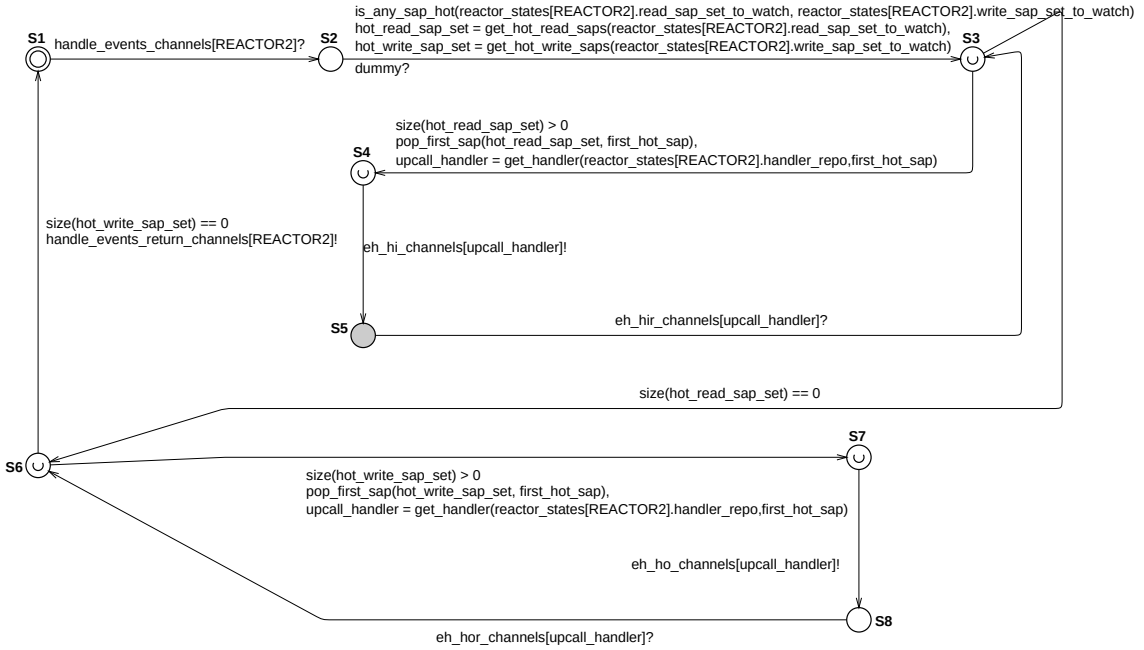
The query E◇ Client.DeadlineMiss is then used to see whether or not there is a deadline miss. The property is satisfied and UPPAAL produces a trace of the sequence of steps as shown in Figure 6.22 that led to the deadline miss. We have cut off the initial part of the trace, which is again just the initialization phase. We start at line 7, when Client1 sends a message through SAP handle 1 to Flow1_EH1. The unidirectional channel transfers this data (lines 8-9) to SAP handle 2, which causes Reactor1 to unblock (line 10) and make an upcall (lines 10-13) to Flow1_EH1, which then performs some computation (l4) and sends a message (line 15) to Flow1_EH2 through handle 3. This message is transferred (lines 16-17) to buffers corresponding to handle 4, which causes Reactor2 to unblock (line 18) and make an upcall (line 19-21) to Flow1_EH2. Flow1_EH2 performs some computation (line 22) and sends a message (23) to Flow1_EH3 on handle 5. The data is transferred (line 24-25) from the write buffer for handle 5 to the read buffer of handle 6. After this the system goes into a state where there are no untimed transitions that are enabled and time advances to 1000, when the Client1 automaton moves to the DeadlineMiss state.

Figures 6.23 and 6.24 show the states of the relevant automata when a deadlock occurs. The UPPAAL simulator identifies this state as a deadlocked state. A deadlocked state is one in which there are no enabled transitions. In this scenario, none of the automata can proceed further from their respective states and hence the composed system is deadlocked. Reactor1_SRHE0 (Figure 6.23(a)) is in state S5 waiting for the upcall to return from Flow1_EH1. Flow1_EH1 (6.24(b)) is waiting in state S3 waiting for a reply back from Flow1_EH2. Reactor2 (Figure 6.23(b))is waiting in state S5 waiting for the upcall return from Flow1_EH2. Flow1_EH2 (Figure 6.24(c))is waiting in state S3 waiting for a reply back from Flow1_EH3. Flow1_EH3(Figure 6.24(d)) is in its start state S1, since Reactor1 never dispatched an upcall to Flow1_EH3 because the reactor is in state S1 and does not check for new I/O events on any SAPs until the upcall to Flow1_EH1 returns. Because of the above deadlock, the client (Figure 6.24(a)) missed its deadline and is in state DeadlineMiss.

We now provide evidence in support of Hypothesis 2.2 with the value of $k = 1$. We increase the number of threads in Reactor1 to 2 by using a TP reactor with 2 threads. Figure 6.25 shows how we instantiate the TP Reactor and the associated thread automata.

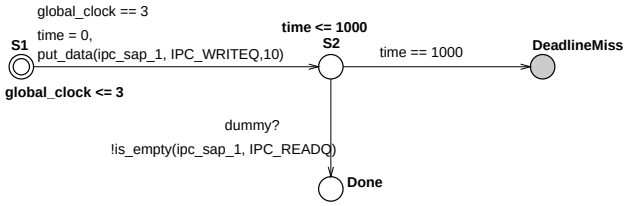


(a) Reactor1

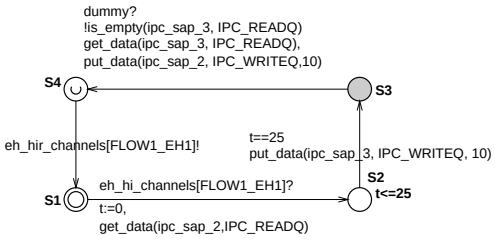


(b) Reactor2

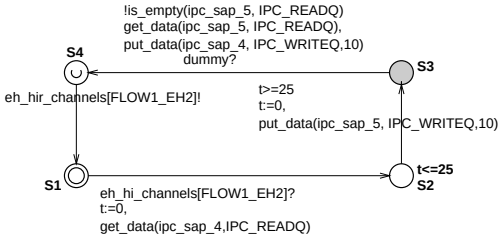
Figure 6.23: Scenario 2 Deadlock in UPPAAL - Reactor Automata States



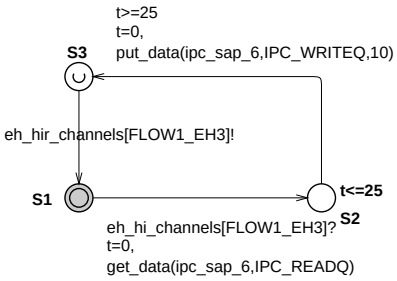
(a) Client1



(b) Flow1_EH1



(c) Flow1_EH2



(d) Flow1_EH3

Figure 6.24: Scenario 2 Deadlock in UPPAAL - Client and Event Handler Automata States

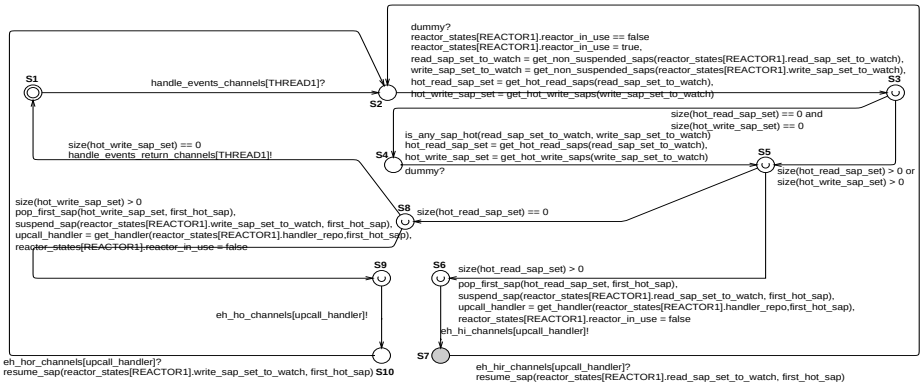
```
Reactor1_TPRHE1 = TP_Reactor(THREAD1,handle_events_channels[THREAD1],
  handle_events_return_channels[THREAD1], reactor_states[REACTOR1]);
Reactor1_TPRHE2 = TP_Reactor(THREAD2,handle_events_channels[THREAD2],
  handle_events_return_channels[THREAD2], reactor_states[REACTOR1]);
Reactor2_TPRHE3 = TP_Reactor(THREAD3,handle_events_channels[THREAD3],
  handle_events_return_channels[THREAD3], reactor_states[REACTOR2]);

Reactor_Thread1 = ReactorThread(handle_events_channels[THREAD1],
  handle_events_return_channels[THREAD1]);
Reactor_Thread2 = ReactorThread(handle_events_channels[THREAD2],
  handle_events_return_channels[THREAD2]);
Reactor_Thread3 = ReactorThread(handle_events_channels[THREAD3],
  handle_events_return_channels[THREAD3]);
```

Figure 6.25: Instantiation in UPPAAL for Scenario 2 with 2 Threads

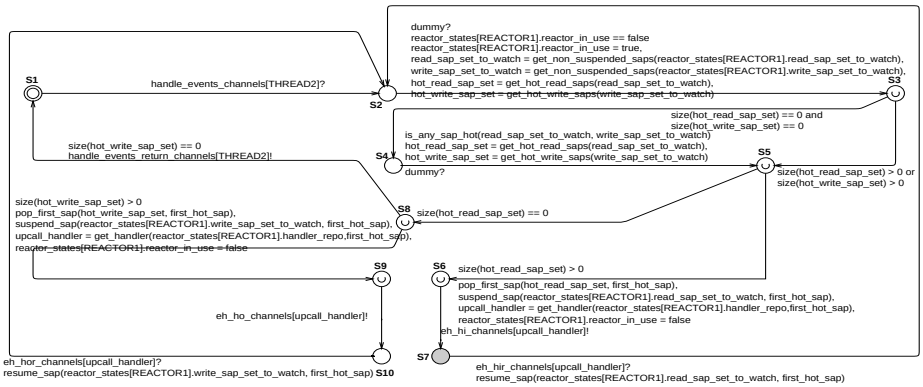
As was discussed in Chapter 4, we instantiate a separate Reactor automaton corresponding to each thread in the thread pool. The reactor states and UPPAAL synchronization channels for communication are passed in as template parameters. The thread automata are then instantiated. The sole function of these threads is to perform a looping handle_events call on the appropriate reactor.

After increasing the number of threads to 2 in Reactor1, the deadlock does not happen. We query whether the Client reaches the Done state in all interleavings using the following query in the UPPAAL verifier - A◇ client1.Done. The UPPAAL verifier executes this query and shows the result that this property is satisfied. Figures 6.26 and 6.27 show the states of the relevant automata when Client1 automaton is in the Done state.

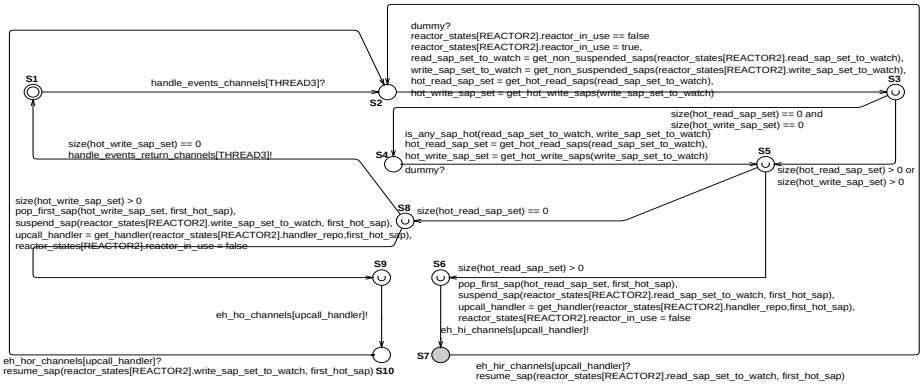
Flow1_EH1 (Figure 6.27(b)) and Flow1_EH2 (Figure 6.27(c)) are in state S4, after each one of them has received replies for their individual request messages. In particular, in state S4, Flow1_EH1 has already completed sending its reply back to Client1. The upcall to Flow1_EH1 was handled by Reactor1_TPRHE1 (Figure 6.26(a)) which is in state S7 waiting for the upcall to return from Flow1_EH1. The upcall to Flow1_EH3 (Figure 6.27(d)) was handled by Reactor1_TPRHE2 (Figure 6.26(b)) which is in state S7 waiting for the upcall to return from Flow1_EH3. Flow1_EH3 (Figure 6.27(d)) is in state S3 after sending its reply back to Flow1_EH2. This system state is in contrast with the deadlocked state that we saw in Figures 6.23 and 6.24.



(a) TP-Reactor1 Thread1

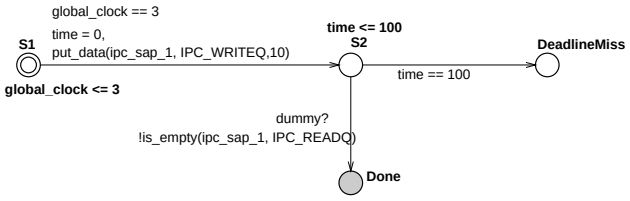


(b) TP-Reactor1 Thread2

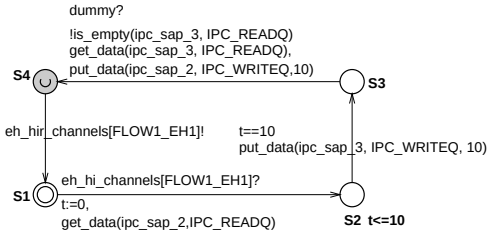


(c) TP-Reactor2 Thread1

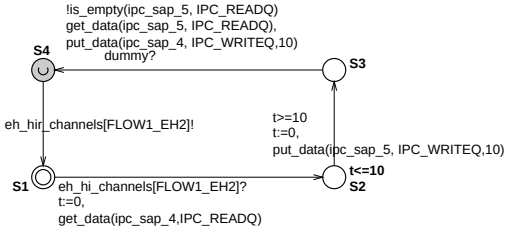
Figure 6.26: Scenario 2 Deadlock in UPPAAL - TP-Reactor Automata States



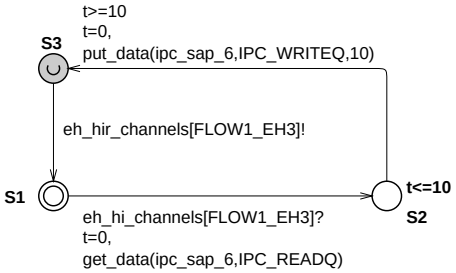
(a) Client1



(b) Flow1_EH1



(c) Flow1_EH2



(d) Flow1_EH3

Figure 6.27: Scenario 2 in UPPAAL with No Deadlock - Client and Event Handler Automata States

Figure 6.28 shows part of an execution trace that led to a state where Client1 is in the Done state. In this trace, as one thread in Reactor1 is making an upcall (line 19) to Flow1_EH1, there is another thread that enters the reactor (line 18) and proceeds to wait on I/O events. This waiting thread then unblocks (lines 34-35) when there is a message waiting on handle 6 for Flow1_EH3. The reactor then makes an upcall (lines 36-37) to Flow1_EH3 thus eliminating the deadlock situation.

```
18   Reactor1_TPRHE1.S2->Reactor1_TPRHE1.S3 { reactor_state.reactor_in_use
18  == 0, dummy?, reactor_state.reactor_in_use := 1, read_sap_set_to_watch
18  := get_non_suspended_saps(reactor_state.read_sap_set_to_watch), write_
18 sap_set_to_watch := get_non_suspended_saps(reactor_state.write_sap_set_
18 to_watch), hot_read_sap_set := get_hot_read_saps(read_sap_set_to_watch)
18 , hot_write_sap_set := get_hot_write_saps(write_sap_set_to_watch) }
19   Reactor1_TPRHE2.S6->Reactor1_TPRHE2.S7 { 1, eh_hi_channels[upcall_han
19 dler]!, 1 }
........
........
34   Reactor1_TPRHE1.S4->Reactor1_TPRHE1.S5 { is_any_sap_hot(read_sap_set_
34 to_watch, write_sap_set_to_watch), dummy?, hot_read_sap_set := get_hot_
34 read_saps(read_sap_set_to_watch), hot_write_sap_set := get_hot_write_sa
34 ps(write_sap_set_to_watch) }
35   Reactor1_TPRHE1.S5->Reactor1_TPRHE1.S6 { size(hot_read_sap_set) > 0,
35 tau, pop_first_sap(hot_read_sap_set, first_hot_sap), suspend_sap(reacto
35 r_state.read_sap_set_to_watch, first_hot_sap), upcall_handler := get_ha
35 ndler(reactor_state.handler_repo, first_hot_sap), reactor_state.reactor
35 _in_use := 0 }
36   Reactor1_TPRHE1.S6->Reactor1_TPRHE1.S7 { 1, eh_hi_channels[upcall_han
36 dler]!, 1 }
37   Flow1_EH3.S1->Flow1_EH3.S2 { 1, eh_hi_channels[eh_pid]?, t := 0, get_
37 data(in_ipc_sap, IPC_READQ) }
```

Figure 6.28: Extracts from UPPAAL Trace Output for Scenario 2 with No Deadlock

6.5.2 Formal Analysis of Scenario 2 in IF

Figure 6.29 shows the execution trace from our IF model of Scenario 2 which was composed from our models of basic ACE building blocks. The trace shows that Client1 sends a message to Flow1_EH1 (lines 3-4). The Select reactor unblocks (line 5) and makes an upcall to Flow1_EH1 (line 6). After the event handler does some computation (line 7), it sends a message to Flow1_EH2 (lines 8-9), which is handled by Reactor2 (line 10) and an upcall is made to Flow1_EH2 (line 11). Flow1_EH2 then

does some further computation (line 12) and sends a request message to Flow1_EH3 (lines 13-14). Since there is no thread in Reactor1 to handle this request, none of the transitions in the system are enabled, which causes time to advance infinitely. Since we want to detect this deadlock situation, we fix the relative deadline of Client1 to be 1000 (which could be any arbitrarily high value) and when this deadline expires, we would infer that there has been a deadlock in the system.

```
3: Client1 : TRACE_SAP_Buffer_Write(1,10)
4: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
5: Reactor1_SRHE0 : TRACE_Reactor_IO_Wait_Done({2,},{})
6: Reactor1_SRHE0 ---handle_input(2)---> Flow1_EH1
7: Time advanced by 25 units. Global time is 28
8: Flow1_EH1 : TRACE_SAP_Buffer_Write(3,10)
9: Unidir_IPC_3_4 : TRACE_SAP_Buffer_Transfer(3,4,10)
10: Reactor2_SRHE0 : TRACE_Reactor_IO_Wait_Done({4,},{})
11: Reactor2_SRHE0 ---handle_input(4)---> Flow1_EH2
12: Time advanced by 25 units. Global time is 53
13: Flow1_EH2 : TRACE_SAP_Buffer_Write(5,10)
14: Unidir_IPC_5_6 : TRACE_SAP_Buffer_Transfer(5,6,10)
15: Time advanced by 951 units. Global time is 1004
16: Client1 : TRACE_DeadlineMiss()
```

Figure 6.29: IF Trace Output for Scenario 2 Leading to Deadlock

We now use two threads in Reactor1 to see whether that eliminates the deadlock. The execution trace for this is shown in Figure 6.30. Note that instead of the Select Reactor we use the Thread Pool Reactor for Reactor1 since there are multiple threads in Reactor1. The trace proceeds exactly as in IF_Trace 5 until Line 14, except that instead of the Select Reactor, it is the TP Reactor that makes the upcall to Flow1_EH1.

After the request message from Flow1_EH2 reaches Reactor1 (line 14), the second thread in the thread pool unblocks from its wait on I/O events (line 15) and makes an upcall to Flow1_EH3 (line 16), which does some computation (line 17) and sends a reply back to Flow1_EH2 (lines 18,20). Flow1_EH2 reads the reply (line 24) and sends its reply back to Flow1_EH1 (lines 25,27). On obtaining this reply Flow1_EH1 reads it (line 31) and sends a reply back to Client1 (line 32,34). This trace shows that with 2 threads, the deadlock in Scenario 2 can be avoided, thus validating Hypothesis 2.2.

```
14: Unidir_IPC_5_6 : TRACE_SAP_Buffer_Transfer(5,6,10)
15: Reactor1_TPRHE2 : TRACE_Reactor_IO_Wait_Done({6,},{})
16: Reactor1_TPRHE2 ---handle_input(6)---> Flow1_EH3
17: Time advanced by 25 units. Global time is 78
18: Flow1_EH3 : TRACE_SAP_Buffer_Write(6,10)
19: Flow1_EH3 ---handle_input_return(0)---> Reactor1_TPRHE2
20: Unidir_IPC_6_5 : TRACE_SAP_Buffer_Transfer(6,5,10)
21: Reactor1_TPRHE2 ---handle_events_return()---> ReactorThread2
22: ReactorThread2 ---handle_events(2)---> Reactor1
23: Reactor1 forks {TP_Reactor_Handle_Events}2
24: Flow1_EH2 : TRACE_SAP_Buffer_Read(5,10)
25: Flow1_EH2 : TRACE_SAP_Buffer_Write(4,10)
26: Flow1_EH2 ---handle_input_return(0)---> Reactor2_SRHE0
27: Unidir_IPC_4_3 : TRACE_SAP_Buffer_Transfer(4,3,10)
28: Reactor2_SRHE0 ---handle_events_return()---> ReactorThread3
29: ReactorThread3 ---handle_events(3)---> Reactor2
30: Reactor2 forks {Select_Reactor_Handle_Events}1
31: Flow1_EH1 : TRACE_SAP_Buffer_Read(3,10)
32: Flow1_EH1 : TRACE_SAP_Buffer_Write(2,10)
33: Flow1_EH1 ---handle_input_return(0)---> Reactor1_TPRHE1
34: Unidir_IPC_2_1 : TRACE_SAP_Buffer_Transfer(2,1,10)
35: Client1 : TRACE_SAP_Buffer_Read(1,10)
```

Figure 6.30: IF Trace Output for Scenario 2 with 2 Threads - No Deadlock

6.6 Scenario 3 – Multiple reactors, *WaitOnReactor* strategy

The problem raised in Scenario 2 by the WaitOnConnection strategy, in which nesting of calls by the single-threaded reactor leads to a deadlock preempted call chain, can be alleviated through use of an alternative strategy for waiting for the reply from the remote event handler, called *WaitOnReactor*. In this case the deadlock is alleviated even with a single thread in the reactor.

Informal Analysis. Figure 6.31 shows the detailed sequence of events that occur on Server1 in Scenario 2 with the WaitOnReactor reply wait strategy when Flow1_EH1 waits for a reply message from Flow1_EH2. First, the handle_events method is called (1) on the reactor by the reactor thread. The reactor waits for I/O events (1.1) to occur on channels corresponding to Flow1_EH1 and Flow1_EH3. On arrival of a request message from Client1, the reactor makes an upcall (1.2) to Flow1_EH1. Flow1_EH1 sends a request message (1.3) to Flow1_EH2. After sending the request

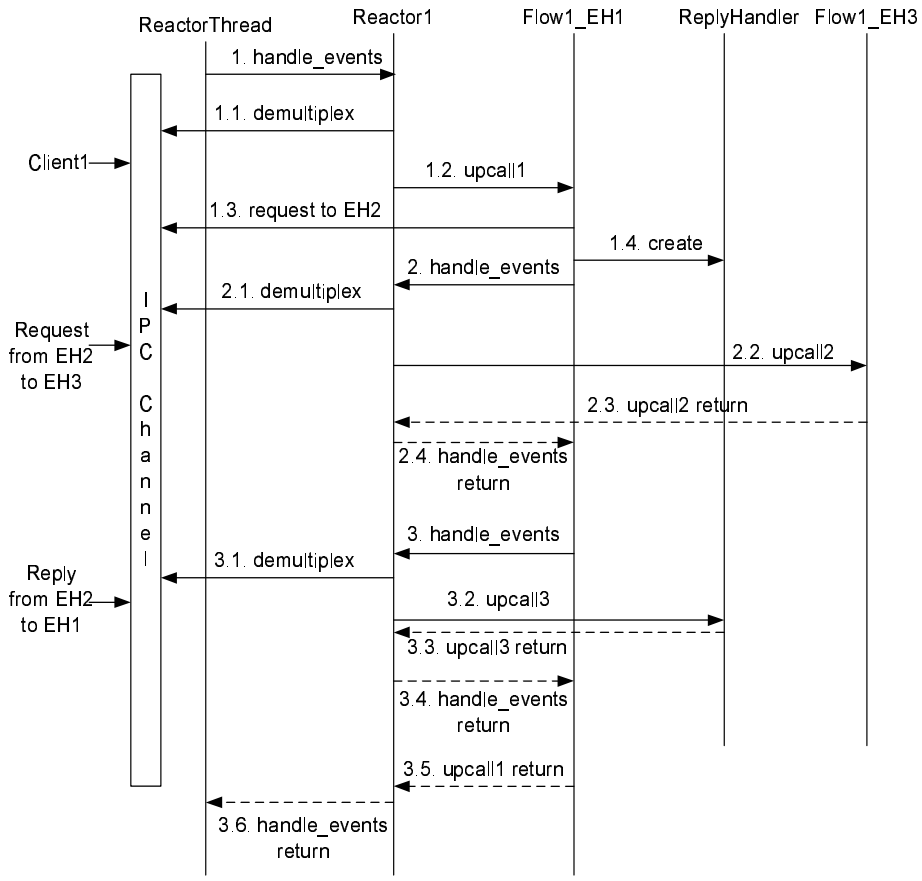


Figure 6.31: Interaction diagram with WaitOnReactor strategy

message, Flow1_EH1 creates (1.4) a ReplyHandler and registers it with the reactor. This ReplyHandler will be the event handler called by the reactor when the reply arrives from Flow1_EH2. After this Flow1_EH1 calls (2) handle_events on the reactor. Note that this is a recursive call to handle_events. The reactor again waits for I/O events (2.1). When the request from Flow1_EH2 that is bound for Flow1_EH3 arrives, the reactor unblocks and makes an upcall (2.2) to Flow1_EH3. After returning from the upcall (2.3), the flow of control goes back to the reactor which in turn hands over control to the caller of handle_events which is Flow1_EH1. Flow1_EH1 now checks with the reply handler to see whether the pending reply has arrived from Flow1_EH2. Since the reply has not arrived, it calls handle_events (3) again on the reactor, which in turn waits for I/O events (3.1). When the reply finally arrives, an upcall is made (3.2) to the ReplyHandler. The ReplyHandler stores the reply and control is returned back (3.3) to the reactor and then to Flow1_EH1 (3.4). Now Flow1_EH1 finds that the pending reply has arrived and the original upcall (upcall1 in Figure 6.31) has returned (3.5) and the first handle_events call (1) is completed (3.6).

From the call sequence it is clear that the deadlock in the previous example is eliminated, since the single thread in Reactor1 is not only waiting on I/O events on a particular SAP handle (handle 3 in Figure 6.18), but rather waits for I/O events on all registered interaction channels (handles 2, 3 and 6 in Figure 6.18).

6.6.1 Formal Modeling of WaitOnReactor in UPPAAL

The model composition in Scenario 3 is very similar to that in Scenario 2 using *WaitOnConnection* and single thread on Reactor1, except for the following differences: (1) instead of a Select_Reactor automaton, as shown in Figure 6.32, we use the Reentrant_Select_Reactor automaton described in Section 4.4.3 and (2) the automaton for Flow1_EH1 is different from the previous scenarios and is shown in Figure 6.33.

The Reentrant_Select_Reactor automaton is used to accommodate the recursive call to handle_events in the same reactor within the context of the same thread. The number of reactor automata that need to be instantiated depends on the depth of the recursive call stack: here we show only two. A new reply event handler is also instantiated to keep track of the reply received by Flow1_EH1 from Flow1_EH2. This

```
Flow1_EH1_RH = ReplyHandler(FLOW1_EH1_RH, 25, ipc_sap_3, REACTOR1);

Reactor1_SRHE0 = Reentrant_Select_Reactor(REACTOR1,
        reentrant_handle_events_channels[REACTOR1][STACK0],
        reentrant_handle_events_return_channels[REACTOR1][STACK0],
        reactor_states[REACTOR1]);

Reactor1_SRHE1 = Reentrant_Select_Reactor(REACTOR1,
        reentrant_handle_events_channels[REACTOR1][STACK1],
        reentrant_handle_events_return_channels[REACTOR1][STACK1],
        reactor_states[REACTOR1]);

Client1 = Client(3, 100, ipc_sap_1);
```

Figure 6.32: Instantiation in UPPAAL for Scenario 2 with WaitOnReactor

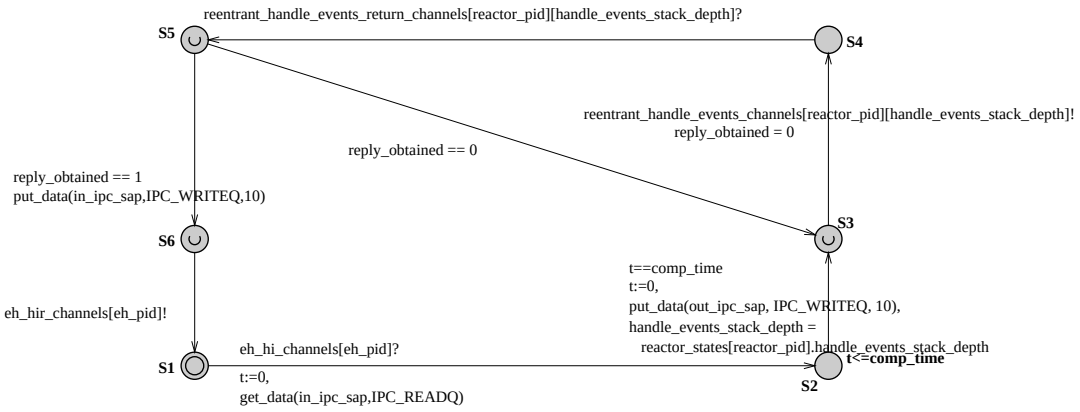


Figure 6.33: Event Handler EH1 Waiting on Reactor for Reply from EH2

reply handler is registered with the reactor to watch ipc_sap_3, which is the SAP through which the reply from Flow1_EH3 arrives at Server1.

After reading the request from the client, Flow1_EH1 reaches state S2. After doing some computation it sends a request to Flow1_EH2 and reaches state S3. Now Flow1_EH1 calls handle_events on the reactor. Note that the current stack depth is stored as part of the reactor's state and this is used to access the appropriate UPPAAL channel to communicate with the appropriate reactor automaton. In state S4, Flow1_EH1 waits for the handle_events call to return. As part of the handle_events processing in the reactor, a reply from Flow1_EH2 to Flow1_EH1 or a request to Flow1_EH3 also could have been processed. On return from the handle_events call in state S5, Flow1_EH1 checks whether the reply for Flow1_EH1 was obtained by the reply handler automaton. If so, it moves to state S6 ending its wait for reply on the reactor. If the reply is still not in, the Flow1_EH1 automaton waits for the reply on the reactor again by moving to state S3.

The following query was verified as true by the UPPAAL verifier - A◇ Client1.Done. This means that there is one state in every path from the start state, where Client1 is in the Done state. The query E◇ Client1.DeadlineMiss was verified as false by the UPPAAL verifier. This means that there is no state in any path from the start state, where Client1 is in the DeadlineMiss state.

```
E<> Client1.Done

17   Flow1_EH1.S3->Flow1_EH1.S4 { 1, reentrant_handle_events_channels[reac
17 tor_pid][handle_events_stack_depth]!, reply_obtained := 0 }
18   Reactor1_SRHE1.S1->Reactor1_SRHE1.S2 { 1, handle_events?, reactor_sta
18 te.handle_events_stack_depth++ }
.......
.......
30   Reactor1_SRHE1.S4->Reactor1_SRHE1.S5 { 1, eh_hi_channels[upcall_handl
30 er]!, 1 }
31   Flow1_EH3.S1->Flow1_EH3.S2 { 1, eh_hi_channels[eh_pid]?, t := 0, get_
31 data(in_ipc_sap, IPC_READQ) }
.......
.......
49   Reactor1_SRHE1.S4->Reactor1_SRHE1.S5 { 1, eh_hi_channels[upcall_handl
49 er]!, 1 }
50   Flow1_EH1_RH.S1->Flow1_EH1_RH.S2 { 1, eh_hi_channels[eh_pid]?, t := 0
50 , get_data(in_ipc_sap, IPC_READQ), reply_obtained := 1 }
```

Figure 6.34: UPPAAL verifyta output for Scenario 2 with WaitOnReactor

Figure 6.34 shows extracts from the trace which leads to an execution state where Client1 is in the Done state. Several things are important to notice in that trace - Flow1_EH1 calls handle_events recursively (line 17) when it is about to wait for reply. This enables the single thread on Reactor1 to wait on both incoming requests and pending replies. Note that the stack depth is incremented (line 18). The upcall to Flow1_EH3 (30-31) is handled in the context of the nested handle_events call stack. Under the context of the same call stack, the upcall to the reply handler (49-50) is also made. This trace shows that our model of the WaitOnReactor reply wait strategy using UPPAAL works as expected. Moreover we have formally shown that the WaitOnReactor reply wait strategy solves the deadlock problem in Scenario 2.

6.6.2 Formal Modeling of WaitOnReactor using IF

Figure 6.35 shows extracts from the IF model of an event handler that uses the WaitOnReactor strategy to wait for reply. To wait for a reply in this manner, a new Reply_Handler is created (A). As part of the creation parameters, the SAP on which to wait for the reply is also passed. This reply handler (not shown) registers itself with the reactor and waits for upcalls based on I/O events on its corresponding SAP. The event handler then calls handle_events (B) on the reactor recursively. This enables the upcall dispatching of events destined for other event handlers including the reply handler. After the return from the nested handle_events call (C), the event handler proceeds to check whether the pending reply has been received. The reply handler on receiving the reply updates one of its state variables indicating that the reply was received. The event handler checks this state variable and on receipt of the pending reply (D) stops calling handle_events. If reply has not been received (E) then the event handler continues to call handle_events recursively.

The execution trace in Figure 6.36 shows the sequence of interactions that take place using the WaitOnReactor strategy. After Reactor1 unblocks from its I/O wait (line 5), it makes the upcall to Flow1_EH1 (line 6), which does some computation (line 7) and sends a request to Flow1_EH2 (lines 8,9). After this, Flow1_EH1 creates a reply handler Flow1_EH1_RH (line 10) which is registered (line 11) with Reactor1 to wait for the reply. Flow1_EH1 then makes a recursive handle_events call (line 14) on

```
state wait_on_reactor;

    reply_handler_ := fork Reply_Handler(context, reactor_,
(A)                       out_sap_, flow_no_, eh_id_);
    nextstate wait_for_reply_handler_ctor_done;
endstate;

state wait_for_reply_handler_ctor_done;
    input ctor_done();
        nextstate do_handle_events;
endstate;

(B) state do_handle_events;
    output handle_events(context, 0) to reactor_;
    nextstate wait_for_handle_events_return;
endstate;

(C) state wait_for_handle_events_return;
    input handle_events_return(par_context);
        nextstate look_for_reply;
endstate;

state look_for_reply #unstable ;
(D)  provided ({Reply_Handler}reply_handler_).reply_obtained_ = 1;
        kill reply_handler_;
        nextstate send_reply;

(E)  provided ({Reply_Handler}reply_handler_).reply_obtained_ = 0;
        nextstate do_handle_events;
endstate;
```

Figure 6.35: Extracts from the IF Model for an Event Handler using WaitOnReactor Reply Wait Strategy

```
3: Client1 : TRACE_SAP_Buffer_Write(1,10)
4: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
5: Reactor1_SRHE0 : TRACE_Reactor_IO_Wait_Done({2,},{})
6: Reactor1_SRHE0 ---handle_input(2)---> Flow1_EH1
7: Time advanced by 25 units. Global time is 28
8: Flow1_EH1 : TRACE_SAP_Buffer_Write(3,10)
9: Unidir_IPC_3_4 : TRACE_SAP_Buffer_Transfer(3,4,10)
10: Flow1_EH1 forks {Reply_Handler}0
11: Flow1_EH1_RH ---register_handler(3,Flow1_EH1_RH,EH_READ,0)--->Reactor1
12: Reactor1 ---register_handler_return()---> Flow1_EH1_RH
13: Flow1_EH1_RH ---ctor_done()---> Flow1_EH1
14: Flow1_EH1 ---handle_events(0)---> Reactor1
15: Reactor1 forks {Select_Reactor_Handle_Events}2
16: Reactor2_SRHE0 : TRACE_Reactor_IO_Wait_Done({4,},{})
17: Reactor2_SRHE0 ---handle_input(4)---> Flow1_EH2
18: Time advanced by 25 units. Global time is 53
19: Flow1_EH2 : TRACE_SAP_Buffer_Write(5,10)
20: Unidir_IPC_5_6 : TRACE_SAP_Buffer_Transfer(5,6,10)
21: Reactor1_SRHE1 : TRACE_Reactor_IO_Wait_Done({6,},{})
22: Reactor1_SRHE1 ---handle_input(6)---> Flow1_EH3
23: Time advanced by 25 units. Global time is 78
24: Flow1_EH3 : TRACE_SAP_Buffer_Write(6,10)
25: Flow1_EH3 ---handle_input_return(0)---> Reactor1_SRHE1
26: Unidir_IPC_6_5 : TRACE_SAP_Buffer_Transfer(6,5,10)
27: Reactor1_SRHE1 ---handle_events_return()---> Flow1_EH1
28: Flow1_EH1 ---handle_events(0)---> Reactor1
29: Reactor1 forks {Select_Reactor_Handle_Events}3
30: Flow1_EH2 : TRACE_SAP_Buffer_Read(5,10)
31: Flow1_EH2 : TRACE_SAP_Buffer_Write(4,10)
32: Flow1_EH2 ---handle_input_return(0)---> Reactor2_SRHE0
33: Unidir_IPC_4_3 : TRACE_SAP_Buffer_Transfer(4,3,10)
34: Reactor2_SRHE0 ---handle_events_return()---> ReactorThread2
35: ReactorThread2 ---handle_events(2)---> Reactor2
36: Reactor2 forks {Select_Reactor_Handle_Events}4
37: Reactor1_SRHE1 : TRACE_Reactor_IO_Wait_Done({3,},{})
38: Reactor1_SRHE1 ---handle_input(3)---> Flow1_EH1_RH
39: Flow1_EH1_RH ---remove_handler(3,{Reply_Handler}0)---> Reactor1
40: Reactor1 ---remove_handler_return()---> Flow1_EH1_RH
41: Flow1_EH1_RH ---handle_input_return(0)---> Reactor1_SRHE1
42: Reactor1_SRHE1 ---handle_events_return()---> Flow1_EH1
43: Flow1_EH1 : TRACE_SAP_Buffer_Write(2,10)
44: Flow1_EH1 ---handle_input_return(0)---> Reactor1_SRHE0
45: Unidir_IPC_2_1 : TRACE_SAP_Buffer_Transfer(2,1,10)
46: Client1 : TRACE_SAP_Buffer_Read(1,10)
```

Figure 6.36: IF Trace Output for Scenario 2 with WaitOnReactor

Reactor1. A new call stack, Reactor1_SRHE1, is created that blocks on I/O events. Note that in the current state of the system, apart from handles 2 and 6, handle 3 is also included in the wait set of Reactor1 since handle 3 is the one on which the reply from Flow1_EH2 to Flow1_EH1 is expected. The sequence of events involving Reactor2 (lines 16-20) is the same as in IF_Trace 5 and 6. Reactor1 now unblocks after waiting on I/O events (line 21). Note that the stack depth of the handle_events call is currently 1 indicated by the 1 at the end of the name Reactor1_SRHE1. The upcall (line 22) to Flow1_EH3 is made which does its computation and sends a reply back (lines 23-26). Now the flow of control is returned back (line 27) to Flow1_EH1 which marks the completion of the recursive handle_events call. Flow1_EH1 now checks with the reply handler to determine whether its reply has arrived yet. Since the reply has not arrived yet, it recursively calls handle_events (line 28) again on Reactor1. Flow1_EH2 gets its reply from Flow1_EH3 and processes the reply and in turn sends (lines 30-33) its own reply back to Flow1_EH1. Reactor1 unblocks and handle 3 is hot (line 37) with the reply message sent from Flow1_EH2. The upcall to the reply handler is made (38) which updates a boolean variable to true indicating that the reply has been obtained, and then unregisters itself from the reactor (39,40). Control returns to Flow1_EH1 after the handle_events call (42). Flow1_EH1 ensures that the reply has been obtained and hence does not call handle_events again. Instead it sends its reply back to Client1 (43,45).

6.6.3 Blocking Factors When Using WaitOnReactor

Even though the deadlock seen in Scenario 2 is eliminated by using the WaitOnReactor strategy, this approach in turn introduces further concurrency issues that must be evaluated. Use of this strategy could, for example, cause blocking delays in the processing of the reply on which Flow1_EH1 is waiting. In Scenario 1, we saw the blocking delay introduced by the reactor because of its serialization behavior. In that case, the blocking delay was because the reactor does not attend to an incoming request until the currently dispatched event handler completes its processing and returns control back to the reactor. In this scenario, we show an example illustrating another kind of blocking factor introduced by recursive upcalls as part of using the WaitOnReactor strategy to wait for replies. The key difference here is that there

is a blocking delay in processing a reply *even after the reactor has dispatched the corresponding event handler (reply handler) that reads the reply.*

The setup for Scenario 3 is shown in Figure 6.37. Here Flow1_EH2 does not depend on any other event handler. Moreover, there is a second flow involving a call from Client2 to Flow2_EH3. In this scenario, as shown in Figure 6.38, Client2 introduces an interleaving call to Flow2_EH3, while Flow1_EH1 is waiting for its reply from EH2. Because of the synchronous nature of the two-way (request-reply) calls made between Client1, Flow1_EH1 and Flow1_EH2, and between Client2 and Flow1_EH3, if the request from Client2 to Flow1_EH3 arrives at almost the same time as the reply from Flow1_EH2, then Flow2_EH3 must finish and send the reply back to Client2 before Flow1_EH1 can start processing the reply from Flow1_EH2.

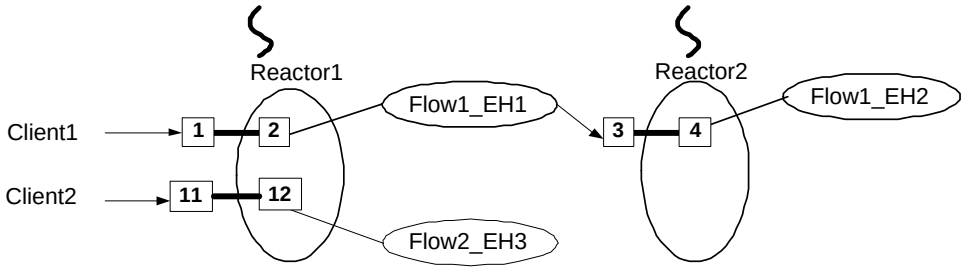


Figure 6.37: Setup for Scenario 3

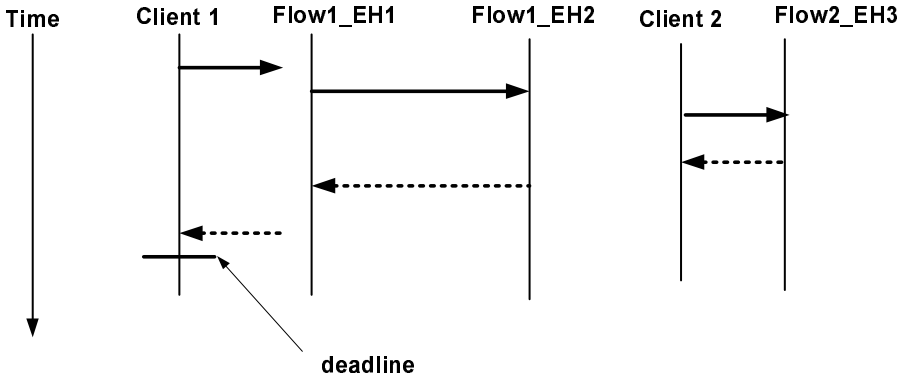


Figure 6.38: Timeline for Scenario 3

Figure 6.39 shows the detailed sequence of events that occurs in Server1. Until step 2.1 the sequence of steps is the same as in Figure 6.31 which we discussed at the beginning of this section. Now let us consider the scenario where both the request from Client2 and the reply from Flow1_EH2 arrive at Reactor1 at about the same time. If the reply is picked up for handling first, then an upcall is made (2.2) to the ReplyHandler, which keeps the reply in a user-space buffer for later processing by Flow1_EH1. At this point (A), the reply is ready for processing by Flow1_EH1, even though it cannot process the reply immediately because the thread of control is still in the reactor which dispatches the next upcall to Flow2_EH3 to handle the request message from Client2. After the upcall (2.4) to Flow2_EH3 returns (2.5), the recursive handle_events finishes, and control returns back (2.6) to Flow1_EH1. Flow1_EH1 now checks (B) with the reply handler to see whether it has received the reply from Flow1_EH2. Since the reply already has been received, Flow1_EH1 can process the reply. Note that the time difference between when A and B occur is considered the blocking delay for the reply from Flow1_EH2 to Flow1_EH1 and this in turn becomes a blocking delay for Client1. It must be noted that there is a distinct difference between this scenario and Scenario 1 as to *where* a pending message is held during the blocking delay. In Scenario 1, while the reactor is processing another upcall, the pending message is held in kernel-level buffers. In this scenario, the reply message is stored in a user-space buffer.

6.6.4 Formal Analysis of WaitOnReactor Blocking Factor in UPPAAL

Figure 6.40 shows extracts from instantiation of the different automata. Note that each event handler has a processing time of 25. The reply event handler does not have any processing time. Note that Client1 sends a request message to Flow1_EH1 at 3 time units from experiment start whereas Client2 sends a request message to Flow2_EH3 at 53 time units. This is exactly the time at which call sequence Flow1_EH1→Flow1_EH2 would have completed and the reply would have been sent to Flow1_EH1.

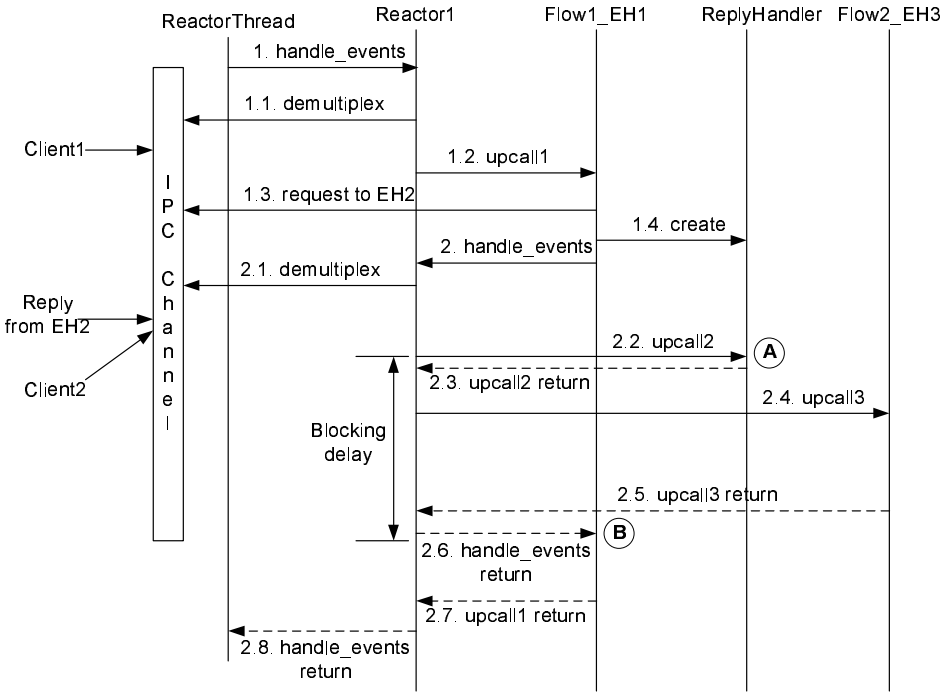


Figure 6.39: Blocking delay with WaitOnReactor

```
Flow1_EH1 = EventHandler1(FLOW1_EH1, 25, ipc_sap_2, ipc_sap_3, REACTOR1);
Flow1_EH2 = EventHandler2(FLOW1_EH2, 25, ipc_sap_4, REACTOR2);
Flow2_EH3 = EventHandler3(FLOW2_EH3, 25, ipc_sap_12, REACTOR1);
Flow1_EH1_RH = ReplyHandler(FLOW1_EH1_RH, 0, ipc_sap_3, REACTOR1);

Client1 = Client(3, 70, ipc_sap_1);
Client2 = Client(53, 100, ipc_sap_11);
```

Figure 6.40: Model Instantiation in UPPAAL for Scenario 3

We used the UPPAAL query E◇ Client1.DeadlineMiss and Flow1_EH1_RH.S4 to see whether Client1 misses its deadline in spite of Flow1_EH1 receiving its reply from Flow1_EH2 on time (indicated by the reply handler Flow1_EH1_RH being in state S4). The verifier finds that there is indeed a deadline miss. Extracts from the trace are seen in Figure 6.41, showing the sequence of events that led to a deadline miss for Client1. The idea here is to see whether the reply handler has already obtained the reply sent from Flow1_EH2 to Flow1_EH1 (state S4 of reply handler Flow1_EH1_RH) and Client1 still misses its deadline. Lines 39-40 show the reactor unblocking and making an upcall when the reply from Flow1_EH2 arrives. Since the reply handler Flow1_EH1_RH is the one that is registered to handle this reply, the upcall is made to that event handler. Flow1_EH1_RH reads the reply from its SAP handle and returns control back to the reactor (41-44). The reactor then proceeds to make the next upcall (46-47) to Flow2_EH3. While Flow2_EH3 is in the middle of processing (48) , Client1 misses its deadline (49).

```
E<> Client1.DeadlineMiss and Flow1_EH1_RH.S4

39   Reactor1_SRHE1.S3->Reactor1_SRHE1.S4 { size(hot_read_sap_set) > 0, ta
39 u, pop_first_sap(hot_read_sap_set, first_hot_sap), upcall_handler := ge
39 t_handler(reactor_state.handler_repo, first_hot_sap) }
40   Reactor1_SRHE1.S4->Reactor1_SRHE1.S5 { 1, eh_hi_channels[upcall_handl
40 er]!, 1 }
41   Flow1_EH1_RH.S1->Flow1_EH1_RH.S2 { 1, eh_hi_channels[eh_pid]?, t := 0
41 , get_data(in_ipc_sap, IPC_READQ), reply_obtained := 1 }
42   Flow1_EH1_RH.S2->Flow1_EH1_RH.S3 { t >= comp_time, tau, t := 0 }
43   Flow1_EH1_RH.S3->Flow1_EH1_RH.S4 { 1, eh_hir_channels[eh_pid]!, 1 }
44   Reactor1_SRHE1.S5->Reactor1_SRHE1.S3 { 1, eh_hir_channels[upcall_hand
44 ler]?, 1 }
45   Reactor1_SRHE1.S3->Reactor1_SRHE1.S4 { size(hot_read_sap_set) > 0, ta
45 u, pop_first_sap(hot_read_sap_set, first_hot_sap), upcall_handler := ge
45 t_handler(reactor_state.handler_repo, first_hot_sap) }
46   Reactor1_SRHE1.S4->Reactor1_SRHE1.S5 { 1, eh_hi_channels[upcall_handl
46 er]!, 1 }
47   Flow2_EH3.S1->Flow2_EH3.S2 { 1, eh_hi_channels[eh_pid]?, t := 0, get_
47 data(in_ipc_sap, IPC_READQ) }
48 Delay: 20
49   Client1.S2->Client1.DeadlineMiss { time == rel_deadline, tau, 1 }
```

Figure 6.41: UPPAAL Trace Output for Scenario 3 Leading to a Deadline Miss

6.6.5 Formal Analysis of WaitOnReactor Blocking Factor in IF

Here we present a trace that shows a deadline miss that results even when the reply from Flow1_EH2 is ready to be processed in a user-space buffer in Server1. We also show a trace for a deadline miss when the reply has arrived at Server1, but has not been detected by Reactor1 yet, and hence it is still in a kernel-space buffer.

Figure 6.42 shows parts of the trace where the reply from Flow1_EH2 is processed and stored in an user-space buffer. The trace extract begins at the the time when both the reply from Flow1_EH2 and the request for Flow2_EH3 have arrived at Server1. The reactor unblocks (line 27) and the two read-ready handles are 12 (corresponding to request from Flow2_EH3) and 3 (corresponding to reply from Flow1_EH2). Reactor1 now makes an upcall to the reply handler Flow1_EH1_RH which stores the reply in a user space buffer. Reactor1 then makes an upcall to Flow2_EH3 which starts computation that goes beyond the deadline for Client1. Since the reply is still in the user buffer and has not been sent to Client1, there is a deadline miss for Client1.

```
27: Reactor1_SRHE1 : TRACE_Reactor_IO_Wait_Done({12,3,},{})
28: Reactor1_SRHE1 ---handle_input(3)---> Flow1_EH1_RH
29: Flow1_EH1_RH ---remove_handler(3,{Reply_Handler}0)---> Reactor1
30: Reactor1 ---remove_handler_return()---> Flow1_EH1_RH
31: Flow1_EH1_RH ---handle_input_return(0)---> Reactor1_SRHE1
32: Reactor1_SRHE1 ---handle_input(12)---> Flow2_EH3
33: Time advanced by 11 units. Global time is 64
34: Client1 : TRACE_DeadlineMiss()
```

Figure 6.42: IF Trace for Scenario 3 Leading to Deadline Miss with Reply in User Buffer

```
21: Reactor1_SRHE1 : TRACE_Reactor_IO_Wait_Done({12,},{})
22: Reactor1_SRHE1 ---handle_input(12)---> Flow2_EH3
23: Flow1_EH2 : TRACE_SAP_Buffer_Write(4,10)
24: Flow1_EH2 ---handle_input_return(0)---> Reactor2_SRHE0
25: Unidir_IPC_4_3 : TRACE_SAP_Buffer_Transfer(4,3,10)
.......
29: Time advanced by 11 units. Global time is 64
30: Client1 : TRACE_DeadlineMiss()
```

Figure 6.43: IF Trace for Scenario 3 Leading to Deadline Miss with Reply in Kernel Buffer

In Figure 6.43, the request from Client2 arrives first at Server1 and the reactor unblocks immediately (line 21). The upcall is made (22-24) and after this the reply from Flow1_EH2 arrives (line 25). This reply is stored in a kernel buffer (in our model this is represented as being stored in the read buffer associated with that SAP). Since the computation of Flow2_EH3 goes beyond the deadline of Client1 (line 29), there is a deadline miss (line 30).

6.7 Scenario 4 – Multiple Reactors, Multiple threads

Informal Analysis. As we saw in Section 6.6, the deadlock scenario in Scenario 2 can be resolved by adding additional reactor threads to Reactor1. However, adding more threads does not guarantee deadlock freedom in general. In this scenario, we consider 3 concurrent flows, each flow representing the sequence of calls in Scenario 2: Client→EH1→EH2→EH3. We use the thread pool reactor in both servers and the thread pools for both reactors have three threads each. Since more than one client might call event handlers on Reactor1 concurrently as shown in Figure 6.44, this could still lead to deadlock as is illustrated in Figure 6.45.

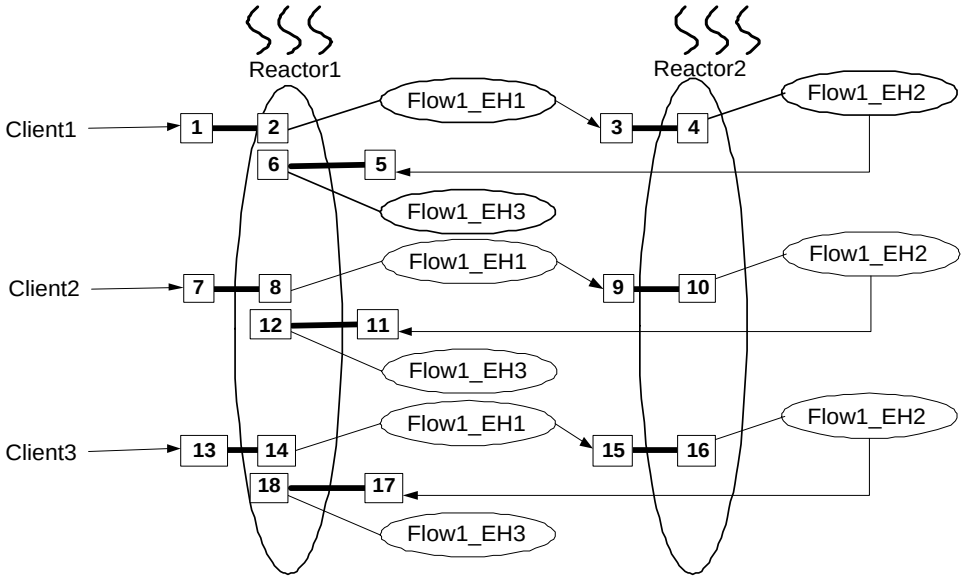


Figure 6.44: Setup for Scenario 4

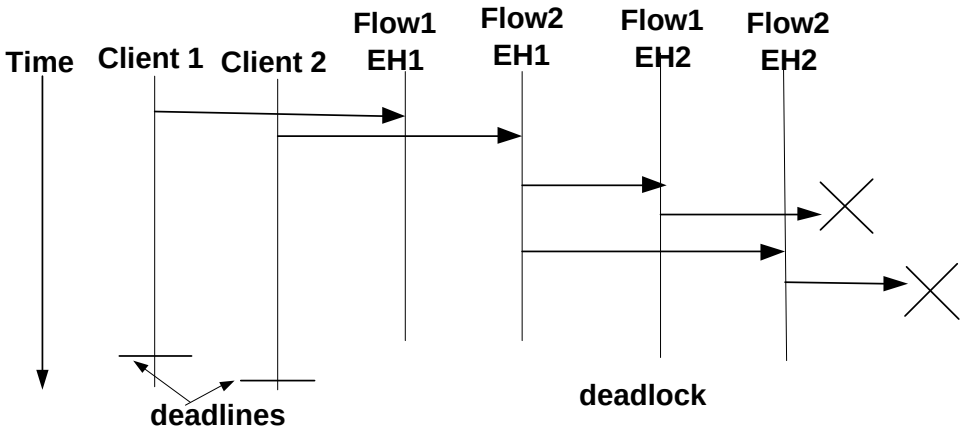


Figure 6.45: Timeline for Scenario 4

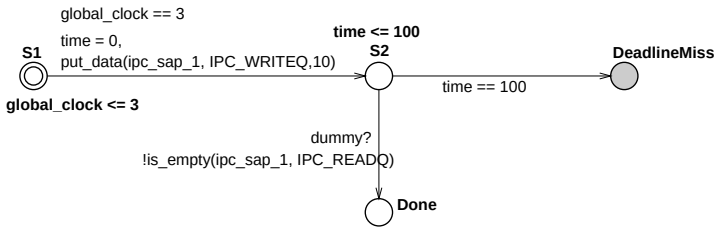
Hypothesis 4. *Any k threads in Reactor1 can be bound by k distinct concurrent calls to the EH1 type event handlers in Reactor1, leaving no threads to handle the call to EH3 type event handlers and thus deadlocking each call chain.*

We provide a more detailed formal analysis of this particular problem, and of alternative protocols to avoid it in [88]. In that research, we developed thread allocation protocols for deadlock avoidance by using the information about the call graph, *e.g.,* the nesting depth at each position of each nested call chain. That work is not a contribution of this dissertation, but we use that protocol as a case study to evaluate the approach presented in this dissertation, as is described in Chapter 8.

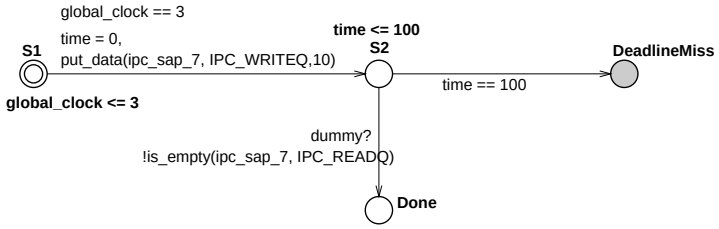
6.7.1 Formal Analysis of Scenario 4 in UPPAAL

We use the following query in the UPPAAL verifier to check whether there are any deadlocks - E◇ deadlock. The verifier finds that the property is satisfied and produces a trace that leads to the deadlocked state. The trace shows a sequence of calls where each flow is blocked at its EH2 instance waiting for a reply from its EH3 instance.

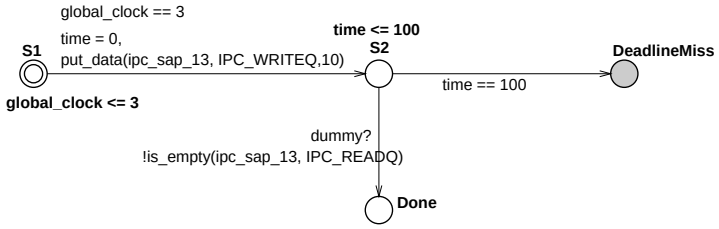
Figure 6.46 shows the three clients in the DeadlineMiss state. Figure 6.47 shows the states of the automaton modeling a TP reactor handle events call. We are showing only two of the 6 TP reactor automata - corresponding to 3 threads in Reactor1 and



(a) Client1

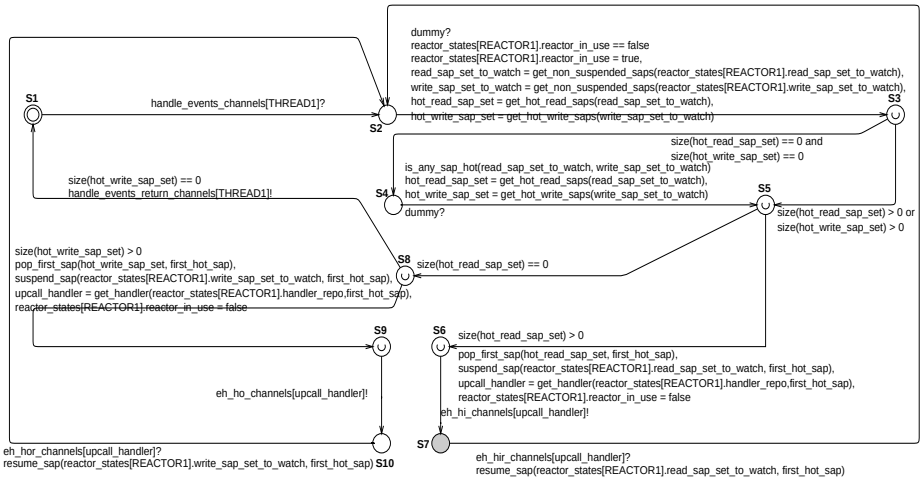


(b) Client2

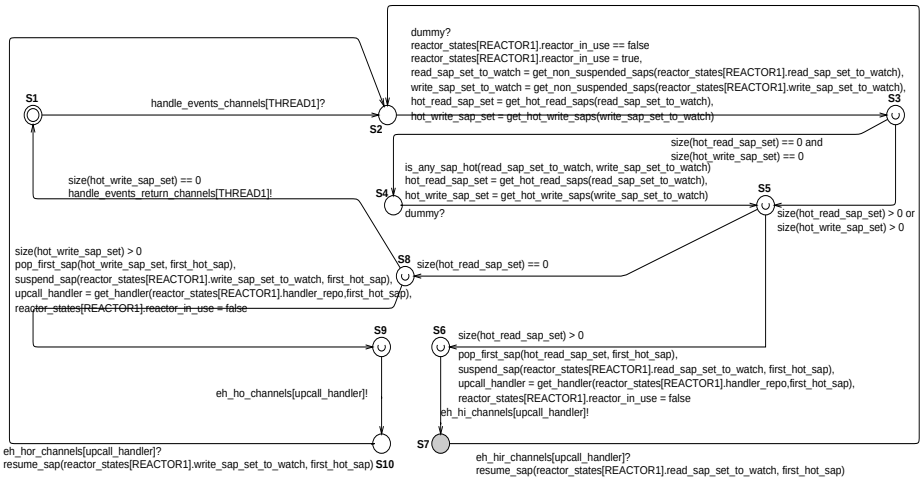


(c) Client3

Figure 6.46: Scenario 4 Deadlock in UPPAAL - Client Automata States

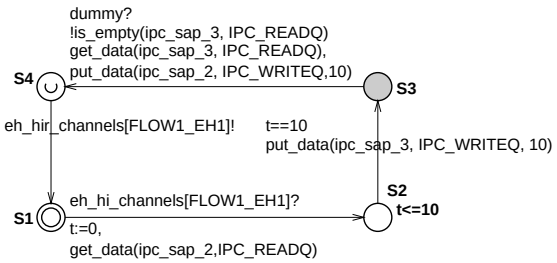


(a) Reactor1 Thread1

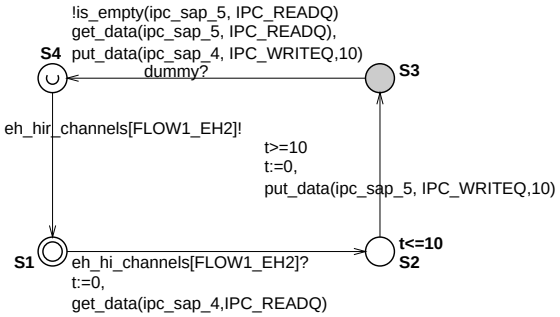


(b) Reactor2 Thread1

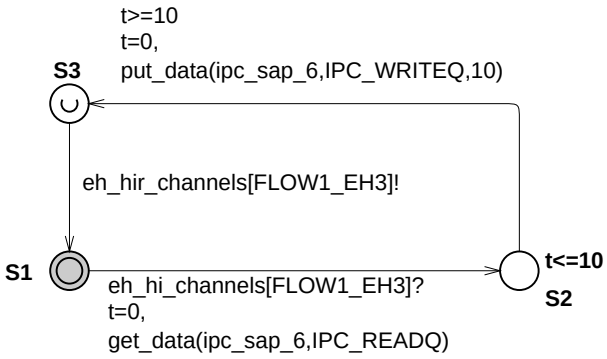
Figure 6.47: Scenario 4 Deadlock in UPPAAL - Reactor Automata States



(a) Flow1_EH1



(b) Flow1_EH2



(c) Flow1_EH3

Figure 6.48: Scenario 4 Deadlock in UPPAAL - Event Handler Automata States

3 threads in Reactor2. Each of these automata was in state S7. Each of the reactor threads was blocked at state S7 waiting for an upcall to finish. The threads in Reactor1 were waiting for the upcalls to the EH1 instances to finish whereas the threads in Reactor2 were waiting for the upcalls to EH2 instances to finish. Figure 6.48 shows the state of the event handlers for Flow1. States for the event handlers in Flow2 and Flow3 are not shown here since they are in similar states as the corresponding event handlers automata for Flow1. The EH1 instances were in state S3 waiting for reply back from their respective EH2 instances. The EH2 instances were waiting for replies back from their respective EH3 instances. However there were no threads left in Reactor1 to dispatch the requests from the EH2 instances destined for the corresponding EH3 instances. This resulted in a deadlocked state where there was no further progress, which resulted in a deadline miss for all of the three clients.

6.7.2 Formal Analysis of Scenario 4 in IF

In this scenario, we show the execution trace to support our informal conjecture that there could be a deadlock with nested upcalls even if we increase the number of threads. If there are concurrent flows that execute the same call sequence, all threads could become exhausted, as the trace in Figure 6.49 shows. The three clients send their request messages to their respective event handlers (lines 3-5). The data transfer is done by the unidirectional channel automata (6-8). The three threads in the thread pool of Reactor1 successively processes the I/O events on handles 2, 8 and 14 (lines 10-14). The event handlers do their respective computations and in turn send requests (16-17, 20-21, 24-25) to their counterparts registered with Reactor2. The three threads in the thread pool of Reactor2 process the three ready handles 4, 10 and 16 (lines 18, 22, 26) and make upcalls to the appropriate handlers (lines 19,23,27). The event handlers in Reactor2 now make further requests to EH3 type handlers on Reactor1 (lines 29-34). There are no reactor threads waiting on Reactor1 to detect these incoming request messages and hence there is a deadlock.

```
1: {Test_Harness}0 ---INIT_MODE_DONE()---> {nil}0
2: Time advanced by 3 units. Global time is 3
3: Client1 : TRACE_SAP_Buffer_Write(1,10)
4: Client2 : TRACE_SAP_Buffer_Write(7,10)
5: Client3 : TRACE_SAP_Buffer_Write(13,10)
6: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
7: Unidir_IPC_7_8 : TRACE_SAP_Buffer_Transfer(7,8,10)
8: Unidir_IPC_13_14 : TRACE_SAP_Buffer_Transfer(13,14,10)
9: Reactor1_TPRHE1 : TRACE_Reactor_IO_Wait_Done({2,8,14,},{})
10: Reactor1_TPRHE1 ---handle_input(2)---> Flow1_EH1
11: Reactor1_TPRHE2 : TRACE_Reactor_IO_Wait_Done({8,14,},{})
12: Reactor1_TPRHE2 ---handle_input(8)---> Flow2_EH1
13: Reactor1_TPRHE3 : TRACE_Reactor_IO_Wait_Done({14,},{})
14: Reactor1_TPRHE3 ---handle_input(14)---> Flow3_EH1
15: Time advanced by 25 units. Global time is 28
16: Flow1_EH1 : TRACE_SAP_Buffer_Write(3,10)
17: Unidir_IPC_3_4 : TRACE_SAP_Buffer_Transfer(3,4,10)
18: Reactor2_TPRHE4 : TRACE_Reactor_IO_Wait_Done({4,},{})
19: Reactor2_TPRHE4 ---handle_input(4)---> Flow1_EH2
20: Flow2_EH1 : TRACE_SAP_Buffer_Write(9,10)
21: Unidir_IPC_9_10 : TRACE_SAP_Buffer_Transfer(9,10,10)
22: Reactor2_TPRHE5 : TRACE_Reactor_IO_Wait_Done({10,},{})
23: Reactor2_TPRHE5 ---handle_input(10)---> Flow2_EH2
24: Flow3_EH1 : TRACE_SAP_Buffer_Write(15,10)
25: Unidir_IPC_15_16 : TRACE_SAP_Buffer_Transfer(15,16,10)
26: Reactor2_TPRHE6 : TRACE_Reactor_IO_Wait_Done({16,},{})
27: Reactor2_TPRHE6 ---handle_input(16)---> Flow3_EH2
28: Time advanced by 25 units. Global time is 53
29: Flow1_EH2 : TRACE_SAP_Buffer_Write(5,10)
30: Unidir_IPC_5_6 : TRACE_SAP_Buffer_Transfer(5,6,10)
31: Flow2_EH2 : TRACE_SAP_Buffer_Write(11,10)
32: Unidir_IPC_11_12 : TRACE_SAP_Buffer_Transfer(11,12,10)
33: Flow3_EH2 : TRACE_SAP_Buffer_Write(17,10)
34: Unidir_IPC_17_18 : TRACE_SAP_Buffer_Transfer(17,18,10)
35: Time advanced by 951 units. Global time is 1004
36: Client1 : TRACE_DeadlineMiss()
```

Figure 6.49: IF Trace for Scenario 4 Leading to Deadlock

6.7.3 Timing Anomaly and Solution

Even though the deadlock was verified in Section 6.7, a subtle timing anomaly in our model was also exposed by the trace in Figure 6.49. In Line 15, we can see that time is advanced by 25 units and in the model the execution of all three EH1 type event handlers in Reactor1 is considered complete. However, the execution of these event handlers on a single-CPU machine would take three times that of a single event handler, assuming that all the three event handlers have the same processing time which is true in this example. This revealed a bug in our model that illustrates the importance of these traces not only to provide clear insights into the interaction between middleware building blocks, but also to help uncover bugs in the models themselves.

The solution to the above problem was to enhance the model to incorporate the notion of a processor as a resource that could be shared among event handlers. This is done using the resource construct in IF. The extract in Figure 6.50 shows how the event handler model is modified to achieve this. When the upcall is made to the event handler, before starting computation the event handler acquires the CPU resource and then releases it after the computation is done.

```
resource CPU;
.....

state do_read;
    acquire CPU;
        ....
        set elapsed := 0;
        nextstate do_compute;
endstate;

state do_compute;
    deadline delayable;
    when elapsed = exec_time_;
        .......
        release CPU;
        nextstate more_service_decision;
endstate;
......
```

Figure 6.50: Model Modifications in IF to Fix Timing Anomaly

The new trace is shown in Figure 6.51. Here each event handler took 25 time units to execute and before the deadlock, the time was 153 (line 36) as opposed to 53 in Figure 6.49.

```
9: Reactor1_TPRHE1 : TRACE_Reactor_IO_Wait_Done({2,8,14,},{})
10: Reactor1_TPRHE1 ---handle_input(2,1)---> Flow1_EH1
11: Reactor1_TPRHE2 : TRACE_Reactor_IO_Wait_Done({8,14,},{})
12: Reactor1_TPRHE2 ---handle_input(8,2)---> Flow2_EH1
13: Reactor1_TPRHE3 : TRACE_Reactor_IO_Wait_Done({14,},{})
14: Reactor1_TPRHE3 ---handle_input(14,3)---> Flow3_EH1
15: Time advanced by 25 units. Global time is 28
16: Flow1_EH1 : TRACE_SAP_Buffer_Write(3,10)
17: Unidir_IPC_3_4 : TRACE_SAP_Buffer_Transfer(3,4,10)
18: Reactor2_TPRHE4 : TRACE_Reactor_IO_Wait_Done({4,},{})
19: Reactor2_TPRHE4 ---handle_input(4,4)---> Flow1_EH2
20: Time advanced by 25 units. Global time is 53
21: Flow1_EH2 : TRACE_SAP_Buffer_Write(5,10)
22: Unidir_IPC_5_6 : TRACE_SAP_Buffer_Transfer(5,6,10)
23: Time advanced by 25 units. Global time is 78
24: Flow2_EH1 : TRACE_SAP_Buffer_Write(9,10)
25: Unidir_IPC_9_10 : TRACE_SAP_Buffer_Transfer(9,10,10)
26: Reactor2_TPRHE5 : TRACE_Reactor_IO_Wait_Done({10,},{})
27: Reactor2_TPRHE5 ---handle_input(10,5)---> Flow2_EH2
28: Time advanced by 25 units. Global time is 103
29: Flow2_EH2 : TRACE_SAP_Buffer_Write(11,10)
30: Unidir_IPC_11_12 : TRACE_SAP_Buffer_Transfer(11,12,10)
31: Time advanced by 25 units. Global time is 128
32: Flow3_EH1 : TRACE_SAP_Buffer_Write(15,10)
33: Unidir_IPC_15_16 : TRACE_SAP_Buffer_Transfer(15,16,10)
34: Reactor2_TPRHE6 : TRACE_Reactor_IO_Wait_Done({16,},{})
35: Reactor2_TPRHE6 ---handle_input(16,6)---> Flow3_EH2
36: Time advanced by 25 units. Global time is 153
37: Flow3_EH2 : TRACE_SAP_Buffer_Write(17,10)
38: Unidir_IPC_17_18 : TRACE_SAP_Buffer_Transfer(17,18,10)
39: Time advanced by 851 units. Global time is 1004
40: Client1 : TRACE_DeadlineMiss()
```

Figure 6.51: IF Trace for Scenario 4 after Timing Anomaly Fix

6.8 Model Checking Costs

In Section 5.2, we described the techniques that we can use to realize the foundational data structures and operations in IF. Here we evaluate the impact of IF mechanisms that we can use to realize the foundational data structures and operations on the cost

of model checking. This evaluation was done in the context of the scenarios described earlier in this chapter. We also evaluated the effect of the state space optimization techniques that we described in Chapter 5, in the context of the models for these scenarios.

6.8.1 Impact of Data Structures in IF

In our models, we have to keep track of different data structures, for example the set of SAP handles, the set of SAP buffers, and the registered handler repository. We implemented these data structures using IF ADTs, arrays, and strings, and evaluated each of these in the context of our models with respect to the time taken and the state space required so that we could pick mechanisms that are most efficient for our models. From the perspective of a model developer, the most convenient mechanism for implementing these is the ADT since it offers a facility to integrate other libraries like STL. Arrays are the least convenient since we have to keep track of the number of elements in the array and write extra code to perform operations like deleting a member from a set. Moreover, while doing interactive exploration, arrays are printed with all slots shown, irrespective of how many elements are actually used in the array, which is an inconvenience when debugging our models. IF strings are dynamic arrays and IF internally keeps track of the number of slots that are occupied and provides operations for insert, remove, *etc.* Strings in IF are not character strings; they are rather dynamic arrays of some IF type.

Table 6.3 shows the number of states, the number of transitions, and the time taken for an exhaustive exploration of the different scenarios using ADTs, IF strings, and arrays. The different scenarios that we evaluated are (1) Scenario 1; (2) Scenario 2 with and without deadlock; (3) Scenario 3 with 1, 2 and 3 flows; and (4) Scenario 4 with deadlock. These exhaustive simulations were run on a Pentium 4 2.8Ghz processor with 1GB RAM. For all the runs, we used the partial order reduction and depth-first-search options in the IF exhaustive simulator, as was suggested by the IF tool developers.

In the course of doing this evaluation, we found that the ADT based implementation is expensive especially when the number of flows increases as is the case in Scenarios

Table 6.3: Impact of data structures in IF on state space

	#1	#2(1)	#2(2)	#3(1)	#3(2)	#3(3)	#4(1)
ADT	124 129 1s	122 124 1s	464 465 1s	1046 1044 1s	99493 99497 193s	656895* 656893* 2724s	17226 17583 52s
Buggy Strings	124 129 1s	122 124 1s	243 247 1s	1057 1058 1s	99493 99497 122s	464190* 464188* 2241s	19581 19751 31s
Strings after bugfix	124 129 1s	122 124 1s	200 204 1s	228 233 1s	936 963 1s	2254 2326 4s	2362 2433 4s
Arrays	124 129 1s	122 124 1s	200 204 1s	228 233 1s	770 795 1s	1652 1708 1s	1897 1966 3s

3(2) and 3(3), where we have 2 flows and 3 flows respectively. Scenario 4 also has three flows, but the state space is not as large as Scenario 3(3) even though the number of flows is the same (3) in both cases. This is because Scenario 4 ends up in a deadlocked state, where there is no further progress, whereas in Scenario 3(3) with 3 flows, there is no deadlock since we use WaitOnReactor strategy and this causes a lot of interleavings that do not arise once a deadlocked state is reached.

Use caution with ADTs. One important observation is that in the context of our models, the ADT based implementation was very expensive when compared to other mechanisms. This was counter-intuitive because ADTs were supposed to be treated as a black box by the IF model checker and the state won't be exposed to the model checker. The original idea was to realize the set of IPC SAP buffers, handler repository, etc using ADTs so that the state of these won't be exposed to the model checker and hence there would be a huge savings in terms of state space. However, our evaluation here shows that this is far from true, which re-emphasizes the need for evaluating such mechanisms in the context of specific models.

The reason for this huge state space with ADTs can be attributed to the manner in which the IF model checker handles ADTs. The value of an abstract data type for the model checker is an uninterpreted bitstring and different bitstrings denote different values. Since our abstract types have values which are C++ STL objects

(vector, map, *etc.*), the same semantic object *e.g.,* a list or map, gets represented twice or more in memory (using different pointers, for instance). These multiple copies are not identified as being equal by IF even if we provide the compare function in the C++ implementation for the ADTs as required by IF. This is because before comparing, the IF model checker takes a blind hash of the value of the type and it compares only the collision lists of the hash key. Different bitstrings gives different hash codes, and this results in the same state being stored multiple times.

For example, assume we have a C/C++ type T. To obtain a hash value on it, IF model checker converts it (via a cast to char*) to a char[n] where n = sizeof(T) and then it takes a sum of its contained characters. IF does not look at the 'semantics' of that type. Instead it uses its internal representation as a char string. Hence two distinct copies of the same list of values *e.g.,* (1,2,3) implemented by using linked lists will lead to different hash codes because the hash code is computed depending on the value of the *pointer* to the next element and not on the *value* of the next element.

With the model using arrays for implementing the foundational data structures, there is a huge savings in state space. We initially thought that using IF strings (dynamic arrays) should not result in a huge increase in state space. But the results were surprising since the state space size was close to that of ADTs. We reported this to the IF developers and they informed us that there was a bug with the IF strings implementation. After we obtained the bug fix for IF strings, we executed our models again and found the state space to be close to those of the array-based models. This again shows the need to evaluate different mechanisms carefully in the context of scenarios like the ones discussed in this chapter, and to ensure that the results are suitable for each specific use case. The IF string based models have state space sizes close to those of the array-based models, and IF strings provide more facilities for the model developer than plain arrays, *e.g.,* insertion and deletion of elements, and printing only the "occupied" slots in an array rather than the whole array. Hence we chose to use IF strings as the basic mechanism for modeling the foundational data structures and operations for our further experiments.

6.8.2 Impact of State Space Optimization in IF

For each scenario, we ran the exhaustive simulation with and without the state space optimization strategies described in Section 5.5. The purpose of this evaluation is to measure the impact on the state space when we apply each of the strategies. We first ran the exhaustive simulation with no priority rules at all. We then incrementally applied each of the following state space optimizations - (1) initialization mode optimization which reduce the number of interleavings during system initialization where the static structure of the system is established; (2) run mode priority rules where we give preference to the client processes over any other process and give the IPC channels preference over all other processes except the client processes; (3) leader/followers ordering rules where the thread with a lower pid gets preference; and (4) run-to-completion rules which reduce number of interleavings by using the concept of a logical thread and each thread running to completion before another thread at the same priority can start running. These techniques are described in detail in Section 5.5. The results are summarized in Table 6.4.

Table 6.4: Impact of State Space Optimization

	#1	#2(1)	#2(2)	#3(1)	#3(2)	#3(3)	#4(1)
	58936	195608	822191*	414427	718701*	583389*	558235*
No prio	215490	889862	2767830*	1692412	2201384*	2271425*	1637672*
rules	38s	185s	2100s*	410s	3360s*	15744s*	6351s*
Init Mode	678	89	7695	1569	75779	531419*	555287*
Prio	1297	98	25917	3786	228219	2081548*	1628964*
Rules	1s	1s	7s	1s	117s	3682s*	5700s
Run Mode	69	82	566	555	4054	14581	11459
Prio	74	87	934	1106	8845	31857	30230
Rules	1s	1s	1s	1s	5s	28s	24s
LF rules	69	82	566	555	4085	16702	2966
	74	87	934	1106	8915	36396	7596
	1s	1s	1s	1s	5s	33s	5s
SchedFifo	124	122	200	228	936	2254	2362
Prio	129	124	204	233	963	2326	2433
Rules	1s	1s	1s	1s	1s	4s	4s

With no state space optimization, the state space is huge. In some cases marked with asterisks, the exploration did not stop, but after around 4 hours, the progress became extremely slow. We reported this problem to the IF developers.

With the introduction of the initialization mode optimization, there was a significant drop in the state space in some cases. For example, in Scenario 1, the number of states went from 58,936 to 678. This shows that the initialization rules are very important, especially since the initialization phase is used to set up only the static relationships in the system and hence does not affect the outcome in any way. However, the initialization rules did not always reduce the state space, for example as was seen in Scenario 4 and Scenario 3(3). This is because these scenarios use multiple flows and thus multiple threads and hence the number of interleavings at simulation time is still huge.

We then added the run mode priority rules. With the run mode priority rules there are ordering optimizations that cause a Client with a lower IF pid number to execute when multiple Client processes become eligible. Since Scenario 1 involves multiple clients, these rules enable some reduction in the state space. In contrast since Scenario 2 does not involve multiple flows, there is not much reduction in the state space. For Scenario 2(2) the state space is reduced since there are multiple threads in Reactor1 and the number of interleavings caused by this is controlled by using ordering optimizations. For Scenarios 3 and 4 also, the state space is reduced because of ordering optimizations. Note that for Scenarios 3 with 3 flows and Scenario 4, the run time ordering optimizations results in significantly reduced state space. This is because the number of flows and hence the number of threads in this scenario are higher.

We then added the leader follower priority rule. With Scenarios 1 and 2 (1), there was no reduction at all. The reason is that these do not have any TP reactors and hence the leader follower priority rule does not apply. Scenario 2(2) does use a TP reactor for reactor1. But there is still no reduction in state space. This is because the choice of which IF process gets access to the reactor is decided in the initialization phase itself and there is no necessity for the leader/follower priority rule to be applied at execution time. Scenario 3(1) has two select reactors and single threads on each of those and hence the L/F prio rule is not applicable and hence there was no reduction in state space. Interestingly, for Scenarios 3(2) and 3(3), there was a slight *increase* in the state space with the addition of the L/F priority rules. These scenarios have only select reactors and hence the L/F prio rules should not apply. With Scenario 4, there are 2 TP reactors and hence the L/F rules reduced the interleavings, resulting in state space reduction.

We then add the run-to-completion rules (SCHED_FIFO rules). With Scenario 1 and 2, there is a slight increase in the state space. This is because these sets of rules have some overhead - (1) maintaining the current logical thread, (2) use of observers to propagate thread ids, and (3) the idle catcher process. As a result, the potential benefits of these rules would be more useful when there is a lot more concurrency in the model or the exploration does not stop because of a deadlock state, as was seen in Scenario 3 where the state space was reduced from 16702 to 2254 by the use of these rules.

We see from these results that the initialization mode ordering optimization reduces the state space more significantly than the other optimization strategies. When concurrency in the system increases as in the number of flows or the number of threads in a reactor, the run mode priority rules, the L/F rules and the run-to-completion rules provide benefits. The results described above highlights the need to evaluate the impact of different state space reduction strategies in the context of representative scenarios.

6.9 Summary

In this chapter we have shown how our models of middleware building blocks can be used to compose models of simple applications and verify their timing and liveness properties using model checking. We demonstrated the utility of our models in formally analyzing different middleware configurations and their effects. We showed how execution traces from a model checker help us pinpoint the reason for the satisfaction or violation of an application property. A set of post-processing tools was discussed, which greatly aids the debugging of models as well as making it more suitable for a human reader. Finally, we presented results on the effective usage of appropriate data structures in IF and the benefits of using the techniques that we developed for state space optimization.

Chapter 7

Model validation

In Chapter 6, we discussed the executable models of representative scenarios that we have developed using UPPAAL and IF. In that chapter we discussed in detail the traces produced by running simulations of these models. We showed the importance of these traces and how they can be extended for post-processing. The discussions about the traces in that chapter were mainly based on domain expertise. In this chapter, we evaluate the fidelity of these models by running actual experiments with these scenarios using the actual ACE building blocks and compare the traces produced by these runs against the ones produced by the simulation. We log events at various points in the kernel, middleware, and application layers using DSUI/DSKI [13] instrumentation points. We then generate a timeline produced by post-processing DSUI/DSKI events and then compare this against a timeline produced by post-processing the output from the corresponding simulation. This establishes a general and reproducible methodology for validation of different middleware configurations using our models.

7.1 Experimental Setup

All the modeling experiments were run using IFx 2.0 (Dec 13, 2005) on a 2.8GHz Pentium 4 with 2GB RAM running RedHat 9 with a 2.4.32 kernel. All empirical experiments were run on a 1.4GHz Pentium 3 with 1GB RAM and running Fedora 2 with a LibeRTOS 2.6.12 kernel.

We enhanced our models to output log events that closely resemble that of the DSUI/DSKI instrumentation points so that we can effectively compare the simulation and actual runs. For each experiment, we identified a set of operating system handles associated with event handlers, and for each of these handles we measured the delay between the time when a message was ready to be read on that handle and the time when an event handler actually started processing that message (handle_input). A socket layer DSKI instrumentation point (EVENT_SOCK_DEF_READABLE DSKI event) was used to log an event whenever a buffer of bytes was enqueued into a socket queue (this instrumentation point is in the *kernel/sock.c: sock_def_readable* function of the Linux 2.6.12 kernel). This function is called by the network-protocol-specific (*e.g.,* TCP/UDP/Unix-sockets) code after enqueueing bytes into a socket queue. This measure increases the accuracy of our measurements when compared to the alternative approach where we might measure the interval between the time when the message was sent by a client and the same message read by an event handler. The problem with the latter approach is that the actual delay may also include the propagation delay of the message, which might skew measurement of the actual blocking delay that is caused by interleaving calls to the same reactor.

To enable correlation between the EVENT_SOCK_DEF_READABLE DSKI event and user space DSUI events, we record the socket handle identifier along with the appropriate DSUI events (*e.g.,* HANDLE_INPUT in an event handler). We also modified the kernel for these experiments to include the socket handle as part of the socket data structure in the kernel so that this information is available during logging of the DSKI event and can be used during post processing to correlate kernel and user events.

For each empirical experiment, we used the DSUI/DSKI event stream and converted it into a timeline. Similarly for each modeling experiment we converted the trace from IF into a timeline. We then compared the two timelines to compare (1) the sequence of events that happened in each of the two cases and (2) the time at which each event happened in each case. In generating both timelines, we start the logical time for the timeline when a client sends a request to an event handler for the first time. For each empirical experiment, we mapped the OS handles to logical handles so that the two timelines would have the same handle numbers for each of the different connections shown in Figure 6.1 in Chapter 6.

7.2 Model Validation for Scenario 1

For Scenario 1, we increased the relative deadline of each client so that there would not be a deadline miss. This helped us in generating a complete timeline without the simulation's exploration getting cut off on a deadline miss by a client. We randomly chose one simulation trace that led to a successful completion (both clients receive replies back before their individual deadlines). The timelines from the simulation (left side) and actual runs (right side) are shown in Figure 7.1.

```
1: 0: BEFORE_CLIENT_SEND_REQUEST(2)
2: 0: EVENT_SOCK_DEF_READABLE(4)
3: 0: BEFORE_CLIENT_SEND_REQUEST(1)
4: 0: EVENT_SOCK_DEF_READABLE(2)
5: 0: HANDLE_INPUT_BEGIN(2)
6: 25: EVENT_SOCK_DEF_READABLE(1)
7: 25: AFTER_CLIENT_RECV_REPLY(1)
8: 25: HANDLE_INPUT_BEGIN(4)
9: 50: EVENT_SOCK_DEF_READABLE(3)
10: 50: AFTER_CLIENT_RECV_REPLY(2)
```

```
1: 0 : BEFORE_CLIENT_SEND_REQUEST(2)
2: 0 : EVENT_SOCK_DEF_READABLE(4)
3: 0 : BEFORE_CLIENT_SEND_REQUEST(1)
4: 0 : EVENT_SOCK_DEF_READABLE(2)
5: 0 : HANDLE_INPUT_BEGIN(2)
6: 25 : EVENT_SOCK_DEF_READABLE(1)
7: 25 : AFTER_CLIENT_RECV_REPLY(1)
8: 25 : HANDLE_INPUT_BEGIN(4)
9: 51 : EVENT_SOCK_DEF_READABLE(3)
10: 51 : AFTER_CLIENT_RECV_REPLY(2)
```

Figure 7.1: Comparison of Timelines - Scenario 1

In the following discussions we use line numbers and L is used to refer to the left side trace and R is used to refer the right side trace. From the two timelines, we can see that the model reflects the actual system closely. Both clients send their requests (L1-4, R1-4) at (logical) time 0. Handles 2 and 4 become read-ready and Reactor1 unblocks and the reactor makes an upcall (L5, R5). A computation is performed for 25ms and after this the event handler sends a reply back (L6-7, R6-7) to the client. An upcall is made for the ready-ready handle 4 (L8-10, R8-10) and a reply is sent back to the appropriate client.

7.2.1 Co-Engineering of Model and Software

The traces produced by the model and actual execution in Figure 7.1 are equivalent with respect to the following - (1) requests from both Client1 and Client2 reach Server1 at the same logical time, (2) the reactor at Server1 sees both handles 2 and 4 as read-ready, (3) the reactor at Server1 sequentially dispatches the requests to the appropriate event handlers, (4) the order of upcalls is the same in both the model and actual execution - the upcall corresponding to handle 2 is dispatched before the

upcall corresponding to handle 4 and (5) the blocking factor for the request in handle 4 is 25 time units in both cases.

Even though the above equivalence between the model and the actual software confirms the qualitative fidelity of our models, the above traces still reveal some important quantitative differences. For example, jitter in the execution time in the actual implementation causes the difference seen in the timeline between the model and actual implementation (see L9,R9). Since logical time in the model execution is controlled by the model checker, there is no jitter in the model execution unless jitter is explicitly introduced in the model.

7.2.2 Blocking Delay

Figure 7.1 shows that some of the requests from clients suffer blocking delays. For example, from the simulation results in Figure 7.1, we see that Client2's request reaches handle 4 (L7) at time 0, but that HANDLE_INPUT_BEGIN is called on the appropriate event handler only at time 25 (L17). The empirical result also reflects this (R10, R22). This blocking delay is caused by interference among flows using the same reactor.

We conducted further experiments to analyze the effect of this blocking delay using empirical results from running experiments with 4 flows. The clients synchronize on a barrier before sending requests and this is repeated. From the results shown in Figure 7.2 it can be seen that the blocking delay at a particular socket handle depends on the number of interleaving calls concurrently passing through the reactor. The blocking delays for the different handles are 0, 25, 50 and 75 ms.

We then used the same number of flows in our model. We modeled a barrier synchronizer and had the clients send their requests after unblocking from the barrier synchronizer. We randomly chose one trace that led to a success state where all clients received their reply back within their deadlines. We then computed blocking delays based on the timelines that we showed earlier.

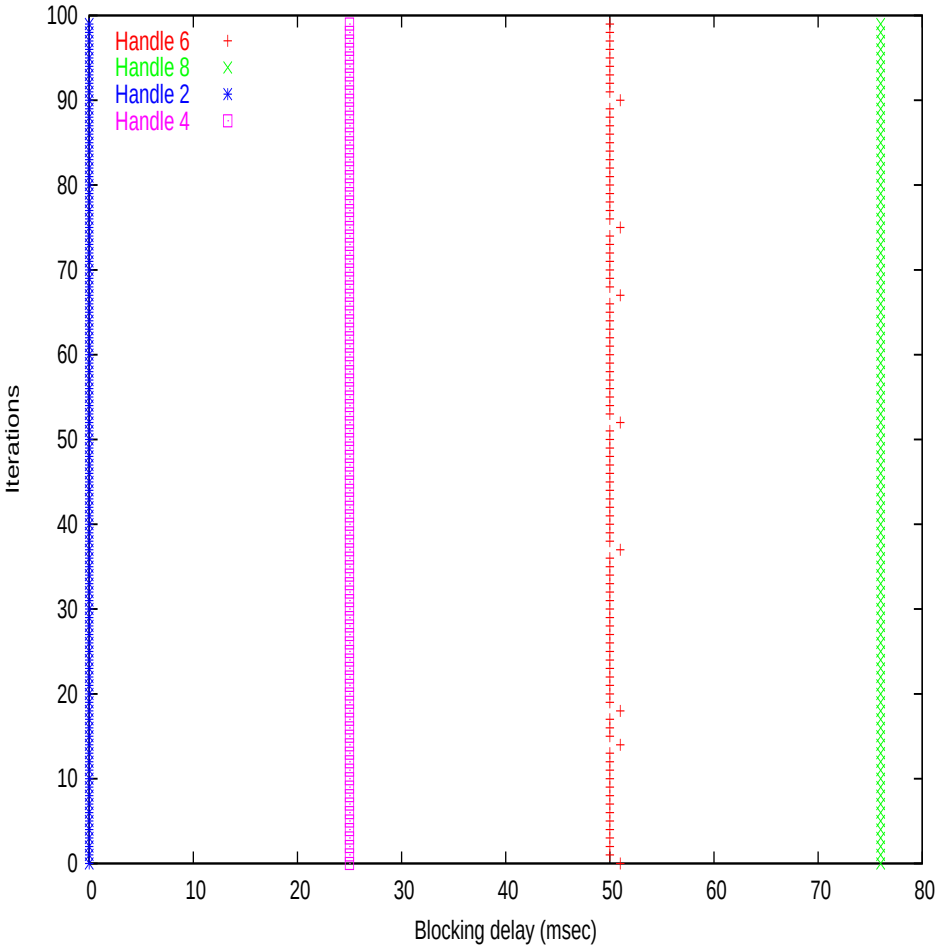


Figure 7.2: Scenario 1 Blocking Factor from Actual Timeline

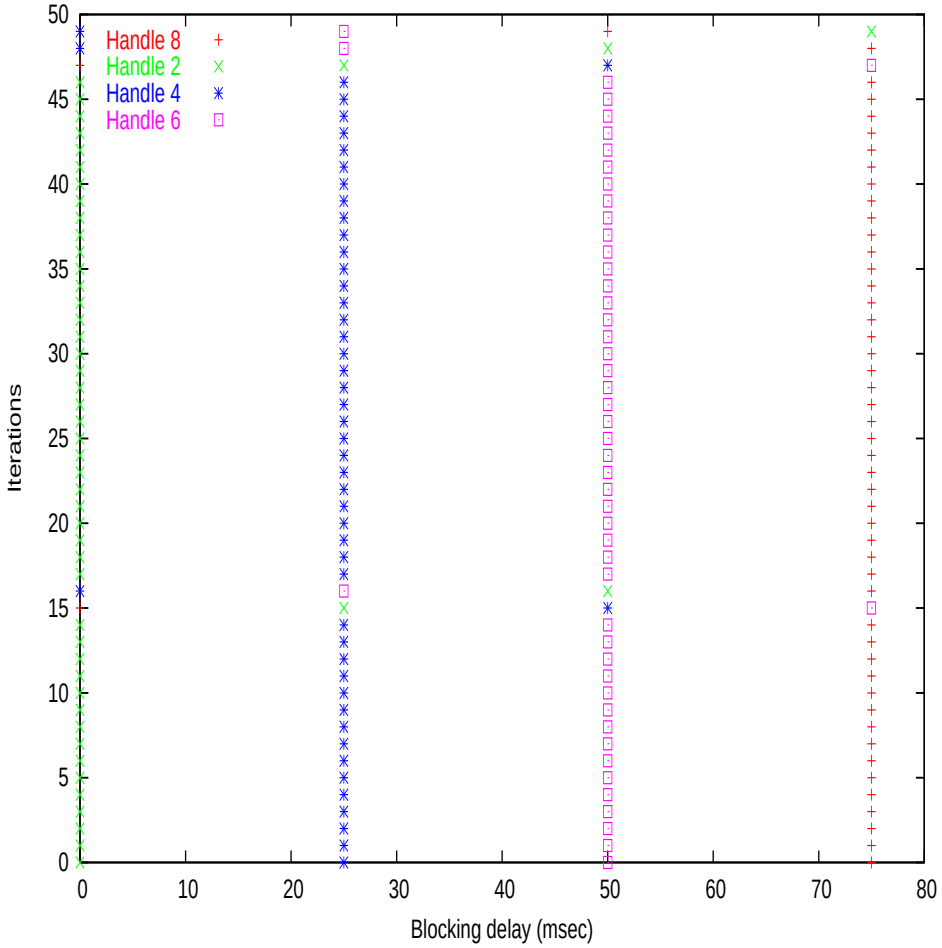


Figure 7.3: Scenario 1 Blocking Factor from Simulation Timeline

From the results shown in Figure 7.3, we can see that the blocking delay varies for a single handle. This shows that the model checker tries out different interleavings as a result of which the same handle suffers different blocking delays because of the arrival patterns of the requests at a reactor and the time when a reactor unblocks.

7.3 Model Validation of Scenario 2

Figure 7.4 compares the simulation and actual timelines for a sequence leading to a deadlock in Scenario 2. In this case according to the timelines, at time 25, after its processing, the event handler sends a request on handle 3 (L4, R4). The request reaches Reactor2 and handle 4 becomes readable (L5, R5). Reactor2 makes an upcall (L6, R6) and the event handler after performing a computation for 25 units sends a request (L7, R7) to Flow1_EH3 which is hosted in Reactor1. Handle 6 becomes ready to read (L8, R8), but the single thread in Reactor1 is blocked on an upcall and hence there is a deadlock.

```
1: 0: BEFORE_CLIENT_SEND_REQUEST(1)
2: 0: EVENT_SOCK_DEF_READABLE(2)
3: 0: HANDLE_INPUT_BEGIN(2)
4: 25: BEFORE_EH_SEND_REQUEST(3)
5: 25: EVENT_SOCK_DEF_READABLE(4)
6: 25: HANDLE_INPUT_BEGIN(4)
7: 50: BEFORE_EH_SEND_REQUEST(5)
8: 50: EVENT_SOCK_DEF_READABLE(6)
```

```
1: 0 : BEFORE_CLIENT_SEND_REQUEST(1)
2: 0 : EVENT_SOCK_DEF_READABLE(2)
3: 0 : HANDLE_INPUT_BEGIN(2)
4: 25 : BEFORE_EH_SEND_REQUEST(3)
5: 25 : EVENT_SOCK_DEF_READABLE(4)
6: 25 : HANDLE_INPUT_BEGIN(4)
7: 51 : BEFORE_EH_SEND_REQUEST(5)
8: 51 : EVENT_SOCK_DEF_READABLE(6)
```

Figure 7.4: Comparison of Timelines - Scenario 2 Deadlock

```
1: 0: BEFORE_CLIENT_SEND_REQUEST(1)
2: 0: EVENT_SOCK_DEF_READABLE(2)
3: 0: HANDLE_INPUT_BEGIN(2)
4: 25: BEFORE_EH_SEND_REQUEST(3)
5: 25: EVENT_SOCK_DEF_READABLE(4)
6: 25: HANDLE_INPUT_BEGIN(4)
7: 50: BEFORE_EH_SEND_REQUEST(5)
8: 50: EVENT_SOCK_DEF_READABLE(6)
9: 50: HANDLE_INPUT_BEGIN(6)
10: 75: EVENT_SOCK_DEF_READABLE(5)
11: 75: AFTER_EH_RECV_REPLY(5)
12: 75: EVENT_SOCK_DEF_READABLE(3)
13: 75: AFTER_EH_RECV_REPLY(3)
14: 75: EVENT_SOCK_DEF_READABLE(1)
15: 75: AFTER_CLIENT_RECV_REPLY(1)
```

```
1: 0 : BEFORE_CLIENT_SEND_REQUEST(1)
2: 0 : EVENT_SOCK_DEF_READABLE(2)
3: 0 : HANDLE_INPUT_BEGIN(2)
4: 26 : BEFORE_EH_SEND_REQUEST(3)
5: 26 : EVENT_SOCK_DEF_READABLE(4)
6: 26 : HANDLE_INPUT_BEGIN(4)
7: 52 : BEFORE_EH_SEND_REQUEST(5)
8: 52 : EVENT_SOCK_DEF_READABLE(6)
9: 52 : HANDLE_INPUT_BEGIN(6)
10: 78 : EVENT_SOCK_DEF_READABLE(5)
11: 78 : AFTER_EH_RECV_REPLY(5)
12: 78 : EVENT_SOCK_DEF_READABLE(3)
13: 78 : AFTER_EH_RECV_REPLY(3)
14: 78 : EVENT_SOCK_DEF_READABLE(1)
15: 78 : AFTER_CLIENT_RECV_REPLY(1)
```

Figure 7.5: Comparison of Timelines - Scenario 2 No Deadlock

We now compare timelines for Scenario2 with a TP reactor for Reactor1 and two threads in the thread pool for Reactor1. Figure 7.5 shows the timelines whose observational equivalence we discuss now. As soon as either handle 2 or handle 6 is ready, the leader thread in Reactor1 unblocks and is ready to dispatch the event handler associated with that handle. Flow1_EH1 now sends a request (L4, R4) that reaches handle 4 (L5, R5). This request is dispatched by Reactor2 (L6, R6) to Flow1_EH2, which in turn sends a request (L7, R7) to Flow1_EH3 on Reactor1 and this makes handle 6 readable (L8, R8). The leader thread in Reactor1 now unblocks and handles the request to Flow1_EH3. In this case also, the jitter in the actual execution causes the timing difference between the actual execution and model execution.

7.4 Model Validation for Scenario 3

```
1: 0: BEFORE_CLIENT_SEND_REQUEST(1)
2: 0: EVENT_SOCK_DEF_READABLE(2)
3: 0: HANDLE_INPUT_BEGIN(2)
4: 25: BEFORE_EH_SEND_REQUEST(3)
5: 25: EVENT_SOCK_DEF_READABLE(4)
6: 25: HANDLE_INPUT_BEGIN(4)
7: 50: BEFORE_EH_SEND_REQUEST(5)
8: 50: EVENT_SOCK_DEF_READABLE(6)
9: 50: HANDLE_INPUT_BEGIN(6)
10: 75: EVENT_SOCK_DEF_READABLE(5)
11: 75: AFTER_EH_RECV_REPLY(5)
12: 75: EVENT_SOCK_DEF_READABLE(3)
13: 75: HANDLE_INPUT_BEGIN(3)
14: 75: AFTER_EH_RECV_REPLY(3)
15: 75: EVENT_SOCK_DEF_READABLE(1)
16: 75: AFTER_CLIENT_RECV_REPLY(1)
```

```
1: 0 : BEFORE_CLIENT_SEND_REQUEST(1)
2: 0 : EVENT_SOCK_DEF_READABLE(2)
3: 0 : HANDLE_INPUT_BEGIN(2)
4: 25 : BEFORE_EH_SEND_REQUEST(3)
5: 25 : EVENT_SOCK_DEF_READABLE(4)
6: 25 : HANDLE_INPUT_BEGIN(4)
7: 51 : BEFORE_EH_SEND_REQUEST(5)
8: 51 : EVENT_SOCK_DEF_READABLE(6)
9: 51 : HANDLE_INPUT_BEGIN(6)
10: 76 : EVENT_SOCK_DEF_READABLE(5)
11: 77 : AFTER_EH_RECV_REPLY(5)
12: 77 : EVENT_SOCK_DEF_READABLE(3)
13: 77 : HANDLE_INPUT_BEGIN(3)
14: 77 : AFTER_EH_RECV_REPLY(3)
15: 77 : EVENT_SOCK_DEF_READABLE(1)
16: 77 : AFTER_CLIENT_RECV_REPLY(1)
```

Figure 7.6: Comparison of Timelines - Scenario 3 No Deadlock

We now proceed to compare timelines for Scenario3 with a Select reactor for Reactor1 and Reactor2 and Flow1_EH1 using the WaitOnReactor strategy to wait for reply. We start our discussion from the point where Flow1_EH1 has completed its computation (L4, R4) and sends a request to Flow1_EH2. After the request is sent, the thread comes back and blocks on the reactor waiting for the reply. After Flow1_EH2 finishes its computation (L7, R7), the request sent to Flow1_EH3 reaches handle 6 (L8, R8). The upcall completes and the reply path continues all the way back to Flow1_EH1. When the reply to Flow1_EH1 arrives (L12, R12) on handle 3, the same thread unblocks to process that reply and makes an upcall to the reply handler (L14,

R14). In this case also, the jitter in the actual execution causes the timing difference between the actual execution and model execution.

7.5 Model Validation for Scenario 4

Figure 7.7 shows a timeline comparison between a model simulation and an actual implementation of Scenario 4. The right side shows an actual execution sequence leading to a deadlock and the left side shows one of the simulation sequences that is similar to the actual execution sequence, which also leads to a deadlock. In the simulation run, handles 2, 8 and 14 are dispatched by Reactor1 one after the other (LR7,8,11) and handles 4, 10 and 16 are dispatched by Reactor2 (LR14,19,22). The event handlers complete their computation in the following sequence - Flow1_EH1 (L9), Flow2_EH1 (L12), Flow1_EH2 (L15), Flow3_EH1 (L17), Flow2_EH2 (L20), Flow3_EH2 (L23). After this the system is deadlocked.

```
1: 0 : BEFORE_CLIENT_SEND_REQUEST(3)
2: 0 : EVENT_SOCK_DEF_READABLE(14)
3: 0 : BEFORE_CLIENT_SEND_REQUEST(1)
4: 0 : EVENT_SOCK_DEF_READABLE(2)
5: 0 : BEFORE_CLIENT_SEND_REQUEST(2)
6: 0 : EVENT_SOCK_DEF_READABLE(8)
7: 1 : HANDLE_INPUT_BEGIN(2)
8: 1 : HANDLE_INPUT_BEGIN(8)
9: 25 : BEFORE_EH_SEND_REQUEST(3)
10: 25 : EVENT_SOCK_DEF_READABLE(4)
11: 25 : HANDLE_INPUT_BEGIN(14)
12: 50 : BEFORE_EH_SEND_REQUEST(9)
13: 50 : EVENT_SOCK_DEF_READABLE(10)
14: 50 : HANDLE_INPUT_BEGIN(4)
15: 75 : BEFORE_EH_SEND_REQUEST(5)
16: 75 : EVENT_SOCK_DEF_READABLE(6)
17: 100 : BEFORE_EH_SEND_REQUEST(15)
18: 100 : EVENT_SOCK_DEF_READABLE(16)
19: 100 : HANDLE_INPUT_BEGIN(10)
20: 125 : BEFORE_EH_SEND_REQUEST(11)
21: 125 : EVENT_SOCK_DEF_READABLE(12)
22: 125 : HANDLE_INPUT_BEGIN(16)
23: 150 : BEFORE_EH_SEND_REQUEST(17)
24: 150 : EVENT_SOCK_DEF_READABLE(18)
```

```
1: 0 : BEFORE_CLIENT_SEND_REQUEST(3)
2: 0 : EVENT_SOCK_DEF_READABLE(14)
3: 0 : BEFORE_CLIENT_SEND_REQUEST(1)
4: 0 : EVENT_SOCK_DEF_READABLE(2)
5: 0 : BEFORE_CLIENT_SEND_REQUEST(2)
6: 0 : EVENT_SOCK_DEF_READABLE(8)
7: 1 : HANDLE_INPUT_BEGIN(2)
8: 1 : HANDLE_INPUT_BEGIN(8)
9: 26 : BEFORE_EH_SEND_REQUEST(3)
10: 26 : EVENT_SOCK_DEF_READABLE(4)
11: 27 : HANDLE_INPUT_BEGIN(14)
12: 52 : BEFORE_EH_SEND_REQUEST(9)
13: 52 : EVENT_SOCK_DEF_READABLE(10)
14: 53 : HANDLE_INPUT_BEGIN(4)
15: 78 : BEFORE_EH_SEND_REQUEST(5)
16: 78 : EVENT_SOCK_DEF_READABLE(6)
17: 104 : BEFORE_EH_SEND_REQUEST(15)
18: 104 : EVENT_SOCK_DEF_READABLE(16)
19: 104 : HANDLE_INPUT_BEGIN(10)
20: 130 : BEFORE_EH_SEND_REQUEST(11)
21: 130 : EVENT_SOCK_DEF_READABLE(12)
22: 130 : HANDLE_INPUT_BEGIN(16)
23: 156 : BEFORE_EH_SEND_REQUEST(17)
24: 156 : EVENT_SOCK_DEF_READABLE(18)
```

Figure 7.7: Comparison of Timelines - Scenario 4 Deadlock

The variation between model and actual execution times is again because of the jitter in computation times of the event handlers.

7.6 Summary

In this chapter, we validated the accuracy of our models by comparing timeline traces from execution of our models to DSKI/DSUI traces from running actual implementations of the different scenarios. We showed that timeline traces from the model execution closely reflect the traces from actual execution, which demonstrated the fidelity of our models.

Chapter 8

Case Study 1 - Deadlock Avoidance Protocol

In Chapter 6 we saw how the WaitOnConnection strategy could cause deadlocks even if we increased the number of threads in the reactor thread pool. In complementary research [88], we have developed thread allocation protocols for deadlock avoidance that exploit information about the application's call graph, *e.g.,* the depth of nesting at each position of each call chain. Although the design and proof of correctness of the protocol itself is not a contribution of this dissertation, we use this protocol as a case study not only to use model checking to verify our implementation of the protocol is sound, but also to do further analysis on the blocking delays introduced by the protocol. The modeling and implementation of the DA protocol, described in Section 8.2, are therefore contributions of this dissertation.

8.1 Overview of Deadlock Avoidance Protocols

Deadlock avoidance protocols are hybrid techniques that use both static call graph analysis and run-time protocol code to prevent deadlocks from happening. Most of these protocols are based on *annotations* of the call-graphs. A call graph in the context of this dissertation is a finite tree with each node containing a 2-tuple consisting of a event handler and a reactor. A node (e, r) describes that the event handler e is dispatched by reactor r. An edge from (e_1, r_1) to (e_2, r_2) denotes that event handler e_1, in the course of its execution may invoke e_2 in reactor r_2.

The outcome of the static call graph analysis is a set of integer *annotations* to each node in a call graph, spanning possibly multiple event handlers registered possibly with multiple reactors. The annotations are maps from nodes in the call-graphs to the natural numbers. These annotations are static in the sense that they do not change during the execution of the system, and are shared among all the threads in a thread pool reactor. The annotations can be chosen using different algorithms [88, 89], which balance efficiency and correctness according to application-specific criteria. Once the annotations are assigned, the run-time implementation of the protocol is used to grant or delay dispatching of events within a reactor, to avoid deadlock.

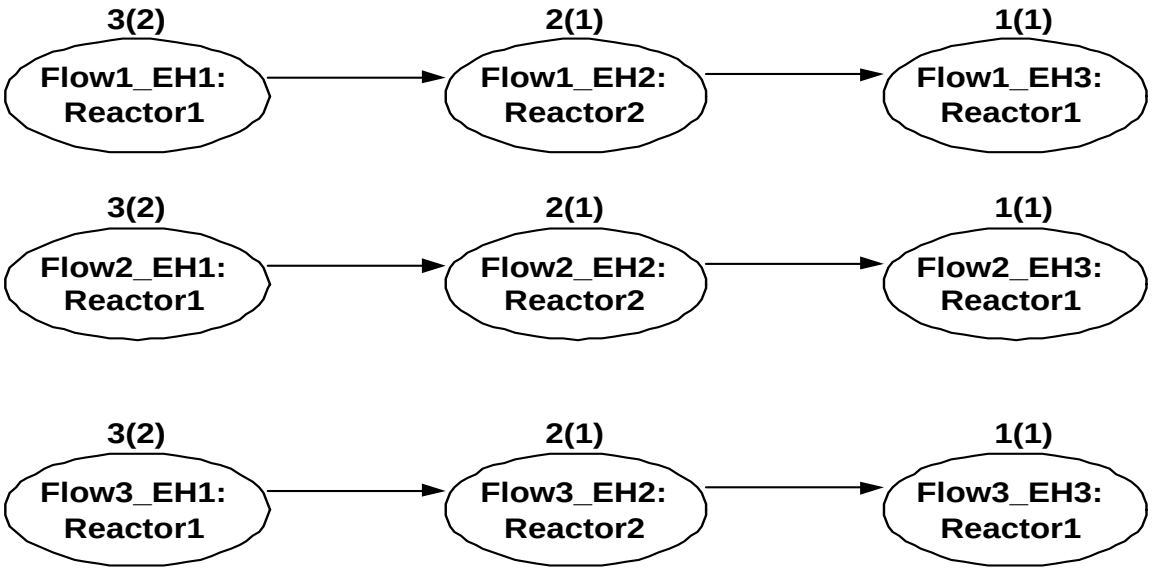


Figure 8.1: Call graph annotations as per DA protocol

Intuitively, the annotation provides a measure of the resources - threads in our case - needed to complete the task corresponding to the node. Two annotations are considered in [88, 89]: *height* and *local height*. Height of a node in a call graph is the usual height of a node in a tree. Local height only takes into account nodes in the same reactor. The local height for a non-leaf node $(e : r)$ is one greater than the maximum of local heights for all descendants of (e, r), which are registered with the same reactor r. For example, Figure 8.1 shows the height annotations and local height height annotations (enclosed in brackets) for Scenario 4.

The run-time aspect of a DA protocol involves executing an entry section before a reactor makes an upcall to an event handler and executing an exit section after the

upcall is completed. A set of state variables is used to keep track of the protocol state. The types of the state variables depend on the specific protocol. One example of such a state variable is the number of available threads - threads that are not doing upcalls to event handlers - in the reactor.

BASIC-P Protocol. The fundamental idea behind protocol BASIC-P is to check whether an incoming call has available all the resources that it can potentially need [88]. A reactor state variable is used to keep track of the threads currently available in the reactor. In the protocol entry section access is granted only if the number of resources indicated by the annotation function at that node is less than or equal to the number of threads available. When access is granted, the number of currently available threads is decremented by one, reflecting that a thread has been allocated.

8.2 Modeling and Implementation of DA Protocols

To model the deadlock avoidance protocol in the context of our thread pool reactor model, we first need to understand the different steps that take place during the delivery of an upcall in a thread pool reactor using the leader/followers pattern. We now describe these steps in the context of the ACE TP reactor and in the process explain how we implemented the deadlock avoidance protocol in the ACE TP reactor so that we could use this implementation to verify the results from our model and validate them with empirical results using this implementation. The DA protocol implementation in the ACE TP reactor, and its modeling and analysis using IF, are additional contributions of this dissertation.

8.2.1 Implementation of DA Protocols

We now describe our implementation of the deadlock avoidance protocol discussed in [88] in the context of the ThreadPool (TP) reactor [97] in ACE [51]. The ACE TP reactor uses the Leader-Followers [97] pattern to share the same reactor instance

among multiple threads in a thread pool. The Leader-Followers pattern has several benefits [97]: (1) it reduces the number of context switches when delivering upcalls since the I/O operation takes place in the same thread context as the event handler upcall; (2) it increases throughput by sharing the workload among multiple worker threads; and (3) it supports long-running service handlers by allowing such a handler to run in the context of one thread in the thread pool while another thread from the same pool waits on the reactor to demultiplex and dispatch other concurrent I/O events.

Figure 8.2 shows how we implemented support for deadlock avoidance (DA) protocols in the context of the ACE TP Reactor, without incurring meaningful overhead for use cases that do not use a DA protocol (as we show empirically in Section 8.2.3) . The additional components that we have introduced to the existing reactor framework in ACE to support DA protocols are shown with a shaded background in Figure 8.2. We now summarize the sequence of events and actions that occur in the TP reactor, and indicate where we have added deadlock avoidance protocol support in this context:

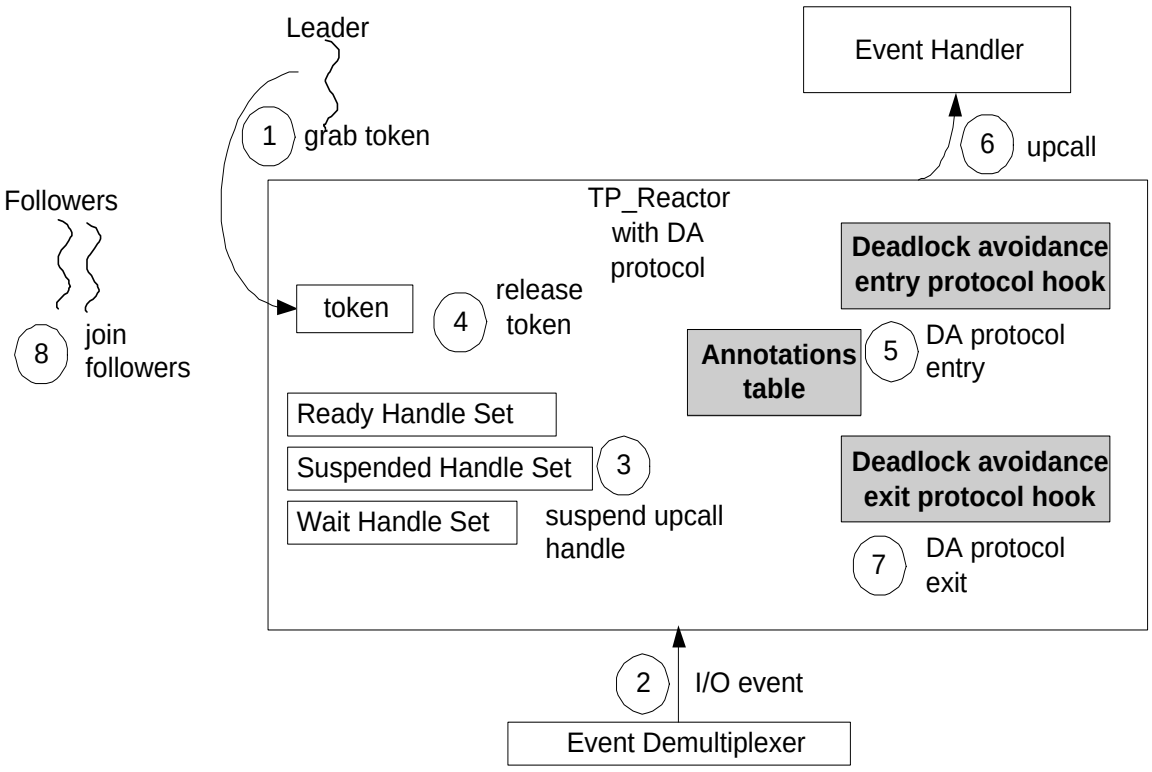


Figure 8.2: Thread Pool Reactor with Deadlock Avoidance

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

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

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

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

5. The former leader thread now executes the deadlock avoidance protocol. In order to impose as few changes to the existing ACE reactor framework as possible, we used the template method design pattern [29] to introduce hook methods before and after the upcall is made. We then added a new class to ACE called Deadlock_Free_TP_Reactor that overrides those hook methods according to the DA protocol. The call graph annotations for the DA protocol are stored within the Deadlock_Free_TP_Reactor, as a table with an annotation for each of the handlers registered with it. The number of available threads in the thread pool is also stored as a state variable in the Deadlock_Free_TP_Reactor. Based on the specific DA protocol, this state variable is incremented and decremented in the protocol's post-upcall and pre-upcall hook methods respectively, and certain I/O handles other than the upcall handle may be suspended. For example in the BASIC-P protocol [88], all handles whose annotations are less than the number of currently available threads in the thread pool are suspended. By default, these hook method implementations are empty methods that can be

inlined out by an optimizing compiler and hence incur little or no overhead as is quantitatively demonstrated in Section 8.2.3.

6. The upcall is made to the event handler.

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

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

8.2.2 Modeling DA Protocol Support using IF

Figure 8.3 shows extracts from the model of a specific DA protocol in the context of the TP reactor model that we discussed in Section 5.3.3. (A) and (B) show the entry and exit protocols respectively. The model shown here implements the BASIC-P protocol [88]. The entry protocol decrements the number of available threads by 1. It then uses the IF procedure disallow_saps to suspend all the SAPs whose call graph annotations are less than the number of available threads. After the upcall to the appropriate event handler, control returns back to the reactor. Based on the return value, the reactor may deregister the handler, resulting in removal of the handler from the handler repository, or it may resume the SAP that was suspended before the upcall. The exit protocol is then executed. The number of available threads is incremented by 1 and disallow_saps is called again to go through the reactor SAP wait set and suspend/resume SAPs based on the new state.

8.2.3 Deadlock Avoidance Protocol Overhead

We conducted empirical evaluations of the overhead of the deadlock avoidance protocol whose implementation was discussed in detail in Section 8.2.1. Our experimental setup is illustrated in Figure 8.4. We used an ACE thread pool reactor watching a set

```
state dispatch_event_handlers;
    provided size(hot_saps_read_set_) > 0;
      next_non_suspended_hot_sap :=
         call ISS_pop_first_non_suspended_sap(hot_saps_read_set_);
      event_handler :=
         call HR_get_handler(({Reactor}reactor_).handler_rep_,
                      next_non_suspended_hot_sap);
      call ISS_suspend_sap(({Reactor}reactor_).sap_read_set_,
                     next_non_suspended_hot_sap);
      output handle_input(context,
                      next_non_suspended_hot_sap) to event_handler;
      task ({Reactor}reactor_).handle_events_in_progress_ :=
         ({Reactor}reactor_).handle_events_in_progress_ - 1;
      task suspended_sap_ := next_non_suspended_hot_sap;
(A)    task ({Reactor}reactor_).avail_threads_ :=
         ({Reactor}reactor_).avail_threads_ - 1;
       call ISS_mark_deny_set(({Reactor}reactor_).sap_read_set_,
                     ({Reactor}reactor_).avail_threads_);
      nextstate wait_for_handle_input_return;

state wait_for_handle_input_return;
    input handle_input_return(par_context, rc);
      call ISS_resume_sap(({Reactor}reactor_).sap_read_set_,
                          suspended_sap_);
(B)   task ({Reactor}reactor_).avail_threads_ :=
              ({Reactor}reactor_).avail_threads_ + 1;
      call ISS_mark_deny_set(({Reactor}reactor_).sap_read_set_,
                     ({Reactor}reactor_).avail_threads_);
      task ({Reactor}reactor_).state_changed_flag_ := 1;
      nextstate done;
endstate;
```

Figure 8.3: Extracts from the IF Model for TP Reactor with Deadlock Avoidance

of ACE_Pipes with only one thread in the thread pool. We used a unique event handler corresponding to each of the pipes. The event handlers do not make any remote function calls, hence the height annotation for each of the event handlers according to the protocol is 1. This is sufficient for measuring the overhead of the protocol implementation within the thread pool reactor because even if the height annotations are different, the mechanisms (the hook functions discussed in Section 8.2.1) for protocol execution are still the same.

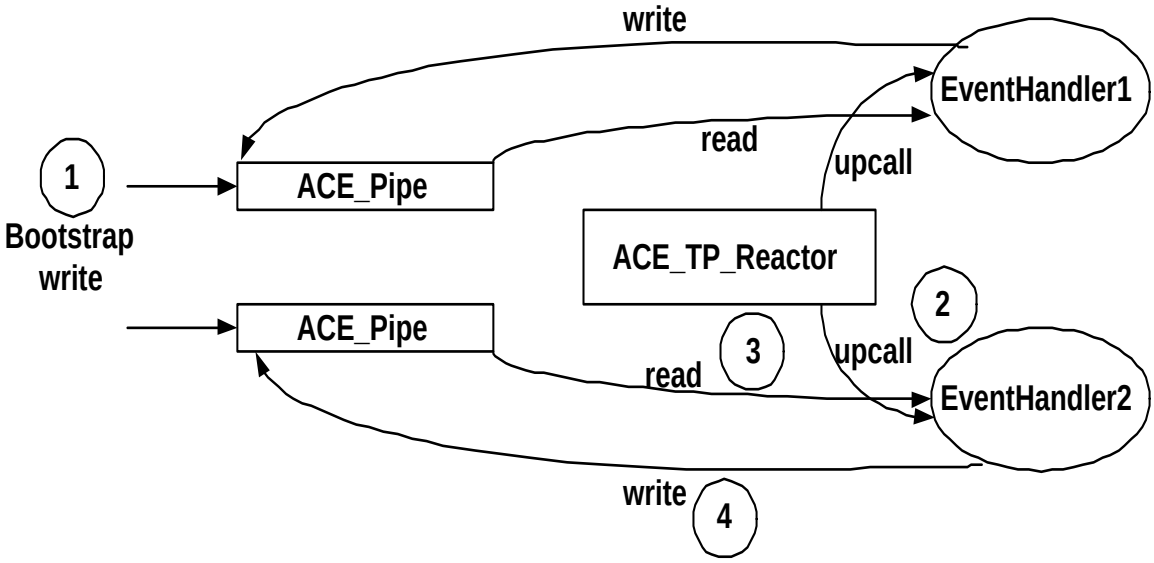


Figure 8.4: DA Protocol Experiment Setup

To bootstrap the experiment, we wrote a constant-sized buffer of bytes to each of the pipes (1). The reactor demultiplexed the events to the appropriate event handlers. On an upcall (2) from the reactor, each event handler read (3) from one end of the pipe and wrote (4) to the other end of the same pipe. In other words, each event handler "fed" itself data. This setup makes it easy to increase the number of event handlers and see the direct effect of that increase on the protocol execution time, since the number of event handlers alone determines the time taken by the reactor to suspend a set of event handlers before making an upcall. We ran these experiments on a Pentium 3 1.4Ghz machine with 1GB RAM. For all of the experiments, we used ACE version 5.4.7, the KUSP Libertos [25] Linux 2.6.12 based kernel, and the Data Streams Kernel Interface (DSKI) [71] and Data Streams User Interface (DSUI) frameworks for instrumentation and processing of collected data.

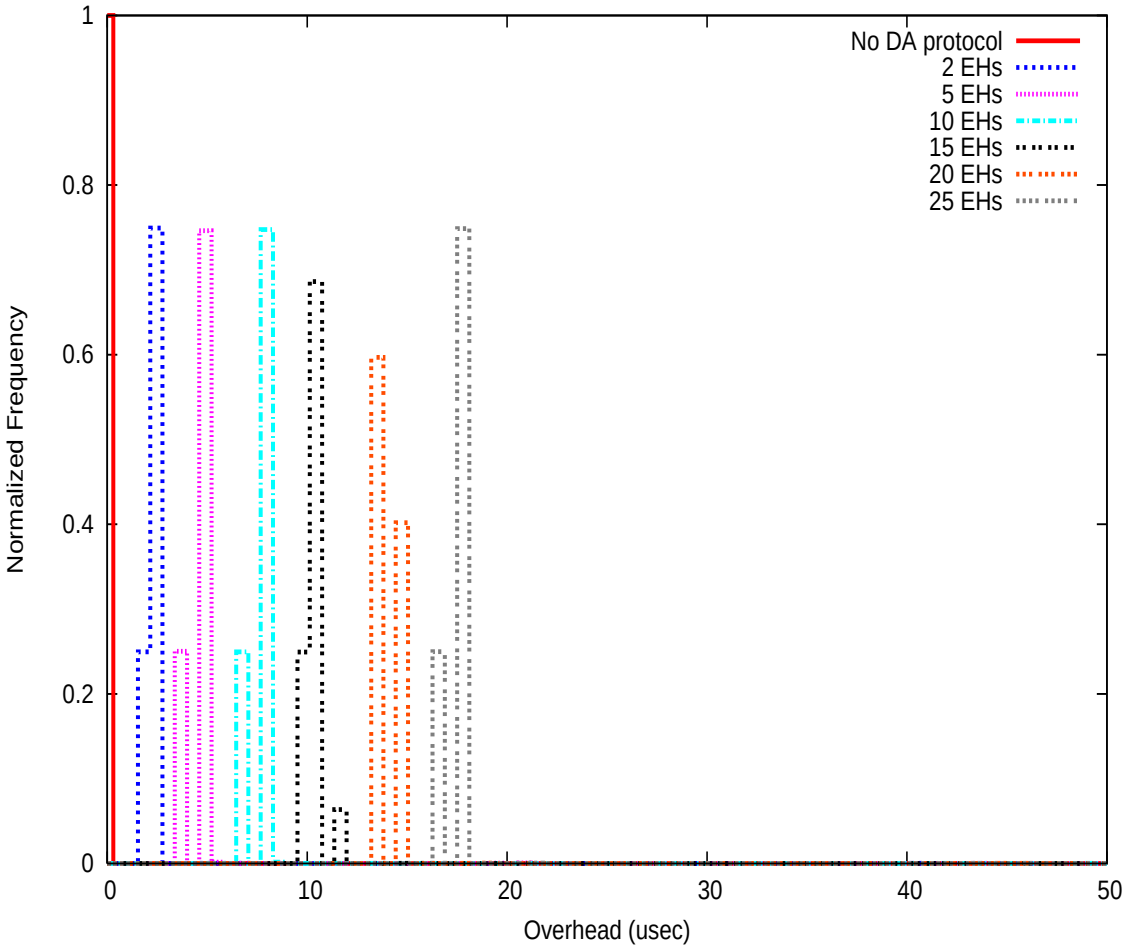


Figure 8.5: DA Protocol Overhead

Figure 8.5 shows the overhead of protocol execution for 2, 5, 15, 20 and 25 event handlers, each with an annotation of 1 and a single thread in the thread pool. As we expected, the time taken was shown to be linear in the number of event handlers, since the protocol implementation suspends all event handlers except one before making an upcall to that event handler. It is significant that without a deadlock avoidance protocol there was no measured overhead, which was our original goal in making the default protocol hook functions empty methods which can be inlined away by an optimizing compiler resulting in little or even no overhead for use cases that don't use a deadlock avoidance protocol.

8.3 Model Checking Deadlock Avoidance Protocols

In this section, we use model checking to verify whether the deadlock avoidance protocol indeed avoids the deadlock seen in Scenario 4 in Chapter 6. We briefly summarize the problem using the trace shown in Figure 8.6.

```
3: Client1 : TRACE_SAP_Buffer_Write(1,10)
4: Client2 : TRACE_SAP_Buffer_Write(7,10)
5: Client3 : TRACE_SAP_Buffer_Write(13,10)
6: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
7: Unidir_IPC_7_8 : TRACE_SAP_Buffer_Transfer(7,8,10)
8: Unidir_IPC_13_14 : TRACE_SAP_Buffer_Transfer(13,14,10)
9: Reactor1_TPRHE1 : TRACE_Reactor_IO_Wait_Done({2,8,14,},{})
10: Reactor1_TPRHE1 ---handle_input(2)---> Flow1_EH1
11: Reactor1_TPRHE2 : TRACE_Reactor_IO_Wait_Done({8,14,},{})
12: Reactor1_TPRHE2 ---handle_input(8)---> Flow2_EH1
13: Reactor1_TPRHE3 : TRACE_Reactor_IO_Wait_Done({14,},{})
14: Reactor1_TPRHE3 ---handle_input(14)---> Flow3_EH1
15: Time advanced by 25 units. Global time is 28
.....
.....
40: Client1 : TRACE_DeadlineMiss()
```

Figure 8.6: IF Trace Showing Deadlock in Scenario 2 with No DA Protocol

All the three clients send their requests at the same time (lines 3-5) and the requests reach Reactor1 at the same time (lines 6-8). The three threads in Reactor1 each make

an upcall to the appropriate event handler (9-14). Since each of the EH1 type event handlers use the WaitOnConnection reply wait strategy, none of these threads is in the reactor waiting on I/O events. As a result any requests from EH2 type event handlers to EH3 type event handlers are not handled, resulting in a deadlock.

8.3.1 Model Verification of DA with BASIC-P

We ran the Scenario 4 model after incorporating the BASIC-P protocol into the TP reactor model. We kept track of the number of available threads in the reactor, and event handlers that had annotations greater than the currently available threads were suspended. To illustrate further the utility of the simulation traces, we now describe another bug in our models that we encountered while running this experiment, and how we fixed it.

Debugging our model using traces. Even after we incorporated the BASIC-P protocol in our model, the model predicted that there would be deadline misses. To find the problem, we added a few more debug IF signals in our model, one of which is TRACE_Reactor_Wait_Set that takes as a parameter the SAP wait set of a reactor. This signal is used to find the set of SAPs that is suspended. Figure 8.7 shows a trace obtained after we added that signal.

In Line 11, notice that the SAP handle 2 is suspended (suspended=1) before making an upcall to Flow1_EH1. With the BASIC-P deadlock avoidance protocol, handles 8 and 14 should also be suspended since their annotations using local heights (3) are greater than the number of currently available threads (2 since one thread is in an upcall) in the reactor. However in Line 11, we notice that the SAP handles 8 and 14 are not suspended. Moreover, in Line 11 we notice that the annotations are 0 for all event handlers. On further inspection, we traced the problem to our test driver where the annotations were never set to those shown in Figure 8.1. Once we set the annotations for the different event handlers, we were able to verify using model checking that there were no deadlocks. The new trace is shown as two parts - one in Figures 8.8 and the other in 8.9.

```
.......
3: Client1 : TRACE_SAP_Buffer_Write(1,10)
4: Client2 : TRACE_SAP_Buffer_Write(7,10)
5: Client3 : TRACE_SAP_Buffer_Write(13,10)
6: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
7: Unidir_IPC_7_8 : TRACE_SAP_Buffer_Transfer(7,8,10)
8: Unidir_IPC_13_14 : TRACE_SAP_Buffer_Transfer(13,14,10)
9: Reactor1_TPRHE1 : TRACE_Reactor_IO_Wait_Done({2,8,14,},{})
10: Reactor1_TPRHE1 ---handle_input(2)---> Flow1_EH1
11: Reactor1_TPRHE1 : TRACE_Reactor_Wait_Set({
    {sap_handle=2,suspended=1,annotation=0},
    {sap_handle=6,suspended=0,annotation=0},{sap_handle=8,suspended=0,
    annotation=0},{sap_handle=12,suspended=0,annotation=0},{sap_handle=14,
    suspended=0,annotation=0},{sap_handle=18,suspended=0,annotation=0},})
12: Reactor1_TPRHE2 : TRACE_Reactor_IO_Wait_Done({8,14,},{})
13: Reactor1_TPRHE2 ---handle_input(8)---> Flow2_EH1
14: Reactor1_TPRHE2 : TRACE_Reactor_Wait_Set({
    {sap_handle=2,suspended=1,annotation=0},
    {sap_handle=6,suspended=0,annotation=0},{sap_handle=8,suspended=1,
    annotation=0},{sap_handle=12,suspended=0,annotation=0},{sap_handle=14,
    suspended=0,annotation=0},{sap_handle=18,suspended=0,annotation=0},})
15: Reactor1_TPRHE3 : TRACE_Reactor_IO_Wait_Done({14,},{})
.......
.......
```

Figure 8.7: IF Trace Revealing a Bug in Our Model

```
3: Client1 : TRACE_SAP_Buffer_Write(1,10)
4: Client2 : TRACE_SAP_Buffer_Write(7,10)
5: Client3 : TRACE_SAP_Buffer_Write(13,10)
6: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,10)
7: Unidir_IPC_7_8 : TRACE_SAP_Buffer_Transfer(7,8,10)
8: Unidir_IPC_13_14 : TRACE_SAP_Buffer_Transfer(13,14,10)
9: Reactor1_TPRHE1 : TRACE_Reactor_IO_Wait_Done({2,8,14,},{})
10: Reactor1_TPRHE1 ---handle_input(2)---> Flow1_EH1
11: Reactor1_TPRHE1 : TRACE_Reactor_Wait_Set({
    {sap_handle=2,suspended=1,annotation=3},
    {sap_handle=6,suspended=0,annotation=1},
    {sap_handle=8,suspended=2,annotation=3},
    {sap_handle=12,suspended=0,annotation=1},
    {sap_handle=14,suspended=2,annotation=3},
    {sap_handle=18,suspended=0,annotation=1},})
12: Time advanced by 25 units. Global time is 28
13: Flow1_EH1 : TRACE_SAP_Buffer_Write(3,10)
14: Unidir_IPC_3_4 : TRACE_SAP_Buffer_Transfer(3,4,10)
15: Reactor2_TPRHE4 : TRACE_Reactor_IO_Wait_Done({4,},{})
16: Reactor2_TPRHE4 ---handle_input(4)---> Flow1_EH2
17: Reactor2_TPRHE4 : TRACE_Reactor_Wait_Set({
    {sap_handle=4,suspended=1,annotation=2},
    {sap_handle=10,suspended=0,annotation=2},
    {sap_handle=16,suspended=0,annotation=2},})
18: Time advanced by 25 units. Global time is 53
19: Flow1_EH2 : TRACE_SAP_Buffer_Write(5,10)
20: Unidir_IPC_5_6 : TRACE_SAP_Buffer_Transfer(5,6,10)
21: Reactor1_TPRHE2 : TRACE_Reactor_IO_Wait_Done({6,},{})
22: Reactor1_TPRHE2 ---handle_input(6)---> Flow1_EH3
23: Reactor1_TPRHE2 : TRACE_Reactor_Wait_Set({
    {sap_handle=2,suspended=1,annotation=3},
    {sap_handle=6,suspended=1,annotation=1},
    {sap_handle=8,suspended=2,annotation=3},
    {sap_handle=12,suspended=0,annotation=1},
    {sap_handle=14,suspended=2,annotation=3},
    {sap_handle=18,suspended=0,annotation=1},})
24: Time advanced by 25 units. Global time is 78
25: Flow1_EH3 : TRACE_SAP_Buffer_Write(6,10)
26: Flow1_EH3 ---handle_input_return(0)---> Reactor1_TPRHE2
27: Unidir_IPC_6_5 : TRACE_SAP_Buffer_Transfer(6,5,10)
28: Reactor1_TPRHE2 : TRACE_Reactor_Wait_Set({
    {sap_handle=2,suspended=1,annotation=3},
    {sap_handle=6,suspended=0,annotation=1},
    {sap_handle=8,suspended=2,annotation=3},
    {sap_handle=12,suspended=0,annotation=1},
    {sap_handle=14,suspended=2,annotation=3},
    {sap_handle=18,suspended=0,annotation=1},})
```

Figure 8.8: IF Trace Showing DA Protocol Avoiding Deadlock - Part 1

```
29: Reactor1_TPRHE2 ---handle_events_return()---> ReactorThread2
30: ReactorThread2 ---handle_events(2)---> Reactor1
31: Reactor1 forks {TP_Reactor_Handle_Events}6
32: Flow1_EH2 : TRACE_SAP_Buffer_Read(5,10)
33: Flow1_EH2 : TRACE_SAP_Buffer_Write(4,10)
34: Flow1_EH2 ---handle_input_return(0)---> Reactor2_TPRHE4
35: Unidir_IPC_4_3 : TRACE_SAP_Buffer_Transfer(4,3,10)
36: Reactor2_TPRHE4 : TRACE_Reactor_Wait_Set({
    {sap_handle=4,suspended=0,annotation=2},
    {sap_handle=10,suspended=0,annotation=2},
    {sap_handle=16,suspended=0,annotation=2},})
37: Reactor2_TPRHE4 ---handle_events_return()---> ReactorThread4
38: ReactorThread4 ---handle_events(4)---> Reactor2
39: Reactor2 forks {TP_Reactor_Handle_Events}7
40: Flow1_EH1 : TRACE_SAP_Buffer_Read(3,10)
41: Flow1_EH1 : TRACE_SAP_Buffer_Write(2,10)
42: Flow1_EH1 ---handle_input_return(0)---> Reactor1_TPRHE1
43: Unidir_IPC_2_1 : TRACE_SAP_Buffer_Transfer(2,1,10)
44: Client1 : TRACE_SAP_Buffer_Read(1,10)
45: Reactor1_TPRHE1 : TRACE_Reactor_Wait_Set({
    {sap_handle=2,suspended=0,annotation=3},
    {sap_handle=6,suspended=0,annotation=1},
    {sap_handle=8,suspended=0,annotation=3},
    {sap_handle=12,suspended=0,annotation=1},
    {sap_handle=14,suspended=0,annotation=3},
    {sap_handle=18,suspended=0,annotation=1},})
46: Reactor1_TPRHE1 ---handle_events_return()---> ReactorThread1
47: ReactorThread1 ---handle_events(1)---> Reactor1
48: Reactor1 forks {TP_Reactor_Handle_Events}8
49: Reactor1_TPRHE3 : TRACE_Reactor_IO_Wait_Done({8,14,},{})
50: Reactor1_TPRHE3 ---handle_input(8)---> Flow2_EH1
51: Reactor1_TPRHE3 : TRACE_Reactor_Wait_Set({
    {sap_handle=2,suspended=2,annotation=3},
    {sap_handle=6,suspended=0,annotation=1},
    {sap_handle=8,suspended=1,annotation=3},
    {sap_handle=12,suspended=0,annotation=1},
    {sap_handle=14,suspended=2,annotation=3},
    {sap_handle=18,suspended=0,annotation=1},})
52: Time advanced by 25 units. Global time is 103
53: Flow2_EH1 : TRACE_SAP_Buffer_Write(9,10)
54: Unidir_IPC_9_10 : TRACE_SAP_Buffer_Transfer(9,10,10)
55: Reactor2_TPRHE5 : TRACE_Reacor_IO_Wait_Done({10,},{})
56: Reactor2_TPRHE5 ---handle_input(10)---> Flow2_EH2
57: Reactor2_TPRHE5 : TRACE_Reactor_Wait_Set({
    {sap_handle=4,suspended=0,annotation=2},
    {sap_handle=10,suspended=1,annotation=2},
    {sap_handle=16,suspended=0,annotation=2},})
58: Time advanced by 25 units. Global time is 128
```

Figure 8.9: IF Trace Showing DA Protocol Avoiding Deadlock - Part 2

All three clients send requests to the corresponding EH1 type event handlers (lines 3-5) and these requests are transferred to the peer SAPs by the unidirectional IPCs (6-8). The leader thread on Reactor1 unblocks (line 9) and we can see that three handles (2, 8 and 14) are read-ready. The leader thread now makes the upcall to handle 2. Because of the BASIC-P protocol code execution, we can now see that in Line 11, the reactor wait set has handles 8 and 14 suspended. This is because the event handlers associated with handles 8 (Flow2_EH1) and 14 (Flow3_EH1) each have an annotation of 3 and this is less than the number of currently available threads (2) in the reactor. The protocol code does not suspend handles 6 and 12 since their annotations each have a value 1, which is less than the number of currently available threads. The protocol code assigns the value of 2 to the suspended state variable whereas the leader thread assigns a value of 1 to the same variable before making an upcall. This helps us in distinguishing between these two cases of suspending a handle. Flow1_EH1 now completes execution (line 12) and makes a further call (line 13-14) to Flow1_EH2. The leader thread on Reactor2 unblocks (line 15) and makes an upcall (line 16) to event handler Flow1_EH2 that is associated with handle 4. Note that in line 17, even after the protocol code runs, none of the other handles are suspended since their annotations (2 for both handles 10 and 16) are not greater than the number of currently available threads which is 2. Flow1_EH2 completes execution (line 18) and makes a further request (lines 19-20) to Flow1_EH3. The current leader thread in Reactor1 now unblocks (line 21) and makes an upcall (line 22) to Flow1_EH3. Since there is still one more thread currently available in Reactor1, handles 12 and 18 are not suspended because their annotations each have the value 1. Flow1_EH3 now completes execution (line 24), sends reply back (line 25) to Flow_EH2 and then the flow of control is returned (line 26) to Reactor1. The protocol exit code runs and line 28 shows the state of the wait set for Reactor1 after this. Handles 8 and 14 still remain suspended because their annotations are still greater than the number of currently available threads and handle 6 that was suspended before the upcall is now resumed. Flow1_EH2 now reads the reply (line 32) from Flow1_EH3, sends its own reply back (line 33) to Flow1_EH1 and returns control (line 34) to Reactor2. After execution of protocol exit code (line 36), none of the handles are resumed since none were suspended (see line 17) by the protocol entry code. Handle 4 that was suspended before the upcall by the leader thread is now (line 36) resumed. Flow1_EH1 receives its reply back (line 41) from Flow1_EH2 and sends its own reply (line 42-44) back

to Client1. The protocol exit code in Reactor1 executes and the wait set is shown in line 45. Handles 8 and 14 are resumed by the protocol exit code and handle 2 is resumed by the reactor thread. A reactor thread on Reactor1 now unblocks (line 49) because handles 8 and 14 are still read-ready. The upcall is made to Flow2_EH1 and the protocol entry code in Reactor1 suspends handles 2 and 14 and the sequence of steps repeats for handles 8 and 14.

We ran a full simulation with 3 flows and observed that there were no deadlocks (90108 states, 93465 transitions, 178 seconds).

8.4 Deadlock Avoidance Blocking Delays

From the simulation traces of successful completion of Scenario 4 without any deadlocks, we picked a trace at random, generated its timeline and calculated the blocking factors on the various input handles. Figure 8.10 shows the blocking delays according to the model.

According to the model, handles 8 and 14 suffer a blocking delay of 75 and 150 respectively. The blocking delay was because of the suspension of these handles by the BASIC-P protocol entry code when an upcall is made.

Figure 8.11 shows the blocking factors from a run of an actual implementation. Handles 8 and 14 suffer blocking delays of approximately 75ms and 150ms respectively. This shows the accuracy of our models in terms of predicting the timing delays suffered due to blocking at the reactor, which in turn was introduced as part of the enforcement of the DA protocol.

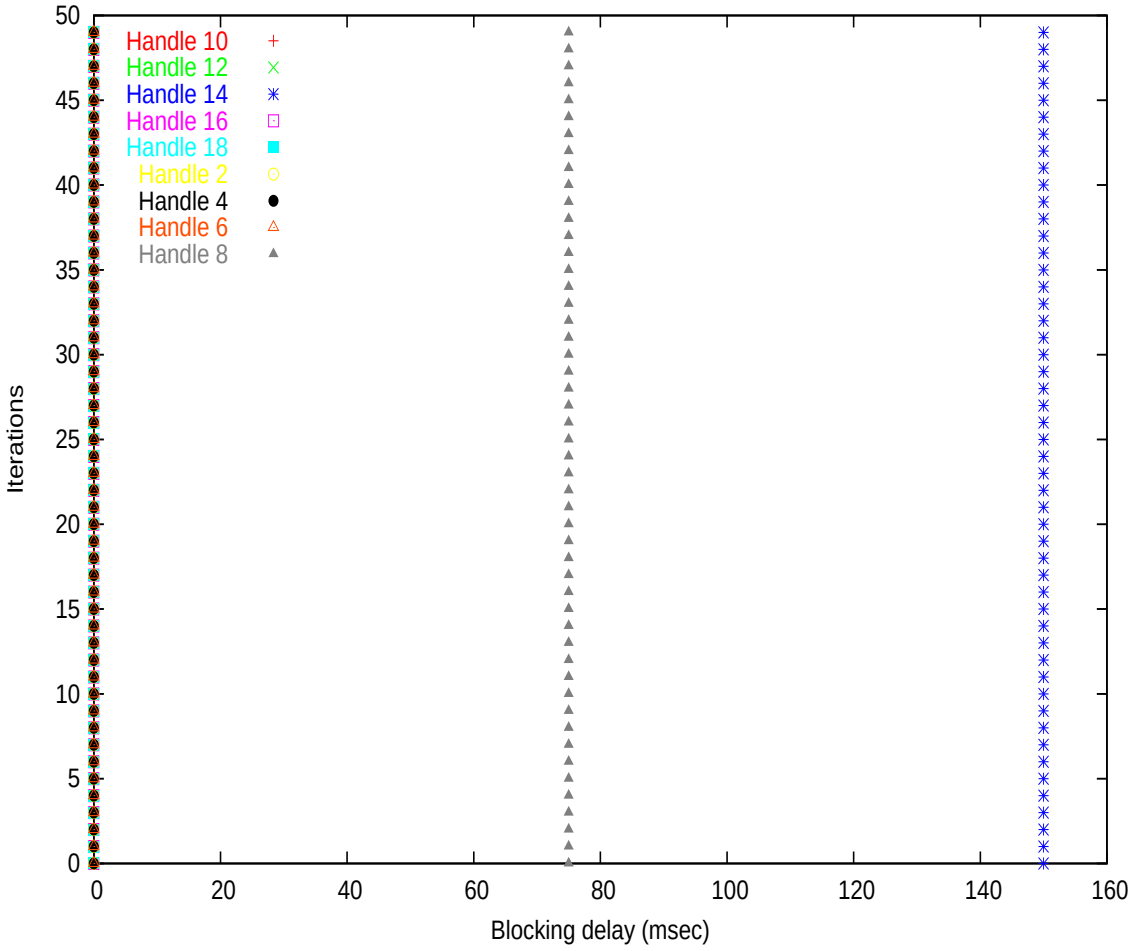


Figure 8.10: Scenario 4 blocking delay Prediction from a Model Execution Trace

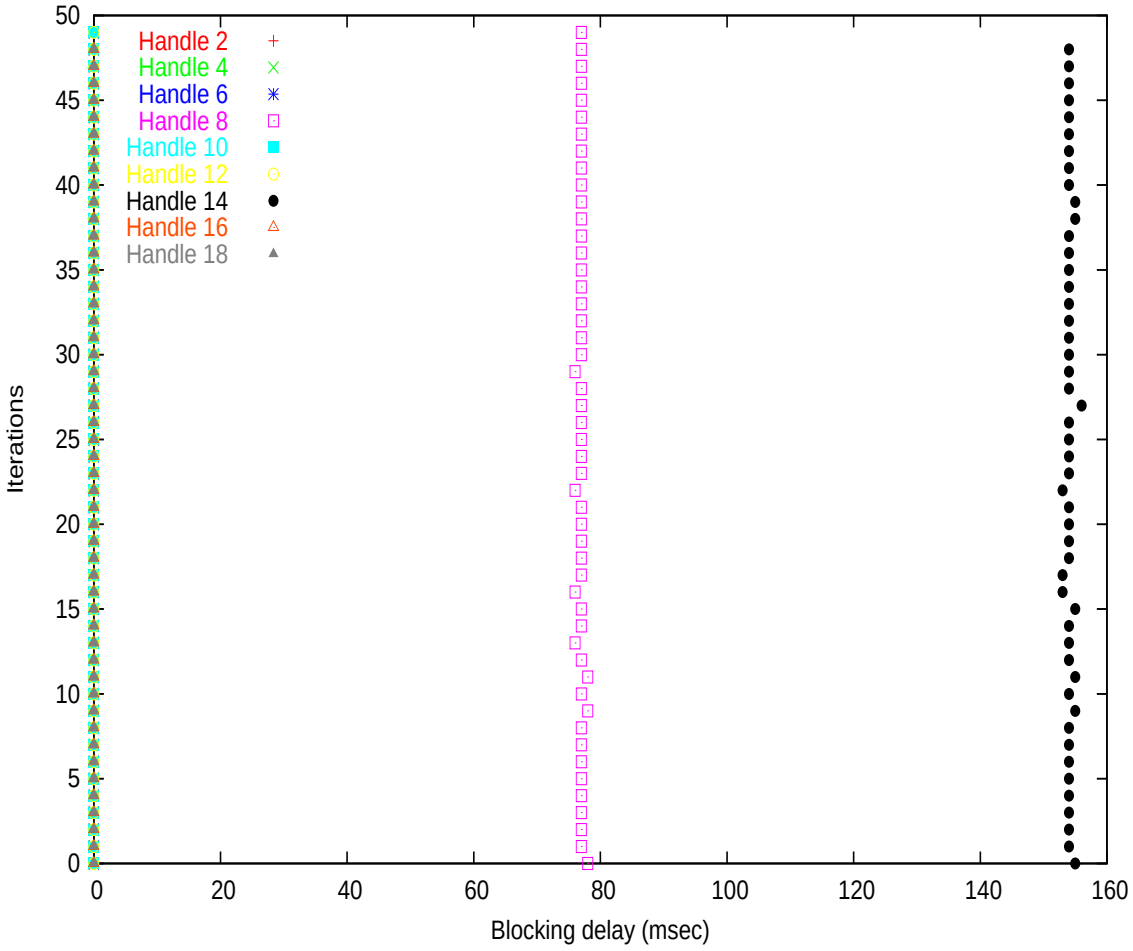


Figure 8.11: Empirical confirmation of Scenario 4 blocking delay

8.5 Summary

In this chapter, we demonstrated the reusability of our models in the context of verification of a deadlock avoidance protocol. We showed how our models uncovered blocking factors introduced by the DA protocol in spite of the success of the protocol in avoiding deadlocks. We described how we implemented the DA protocol in the context of the ACE TP Reactor and empirically measured the performance overhead of the protocol. We observed that our implementation introduces negligible overhead for use cases which do not use the DA protocol.

Chapter 9

Case Study 2 - Application Gateway

While the previous chapters dealt with examples that we created to illustrate the benefits of our modeling techniques and their applicability to new mechanisms like the DA protocol reactor, in this chapter we model an existing example that is distributed with the ACE framework. This case study serves the following purposes - (1) it illustrates the reusability of our models and (2) it serves as an illustrative example of realistic systems[1], where we identify a gap between high level models and actual design and implementation of systems. We illustrate how our low-level models detect the two forms of interference that we discussed in Chapter 1. We show that these forms of interference cannot be captured by high-level system models such as RMA and DREAM and show how middleware-level modeling helps us evaluate different middleware-level design alternatives in the presence of these forms of interference. Finally, we compare the predictions made by our models with empirical results from actual execution of the system.

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

9.1 Overview of Application-level Gateway

We provide a brief overview of the gateway example here, and the reader is referred to [93, 94] for a complete description of this example. The underlying idea of a gateway is the mediator pattern [29] that allows cooperating peers to interact without having to know about each other. A peer that takes the role of a publisher publishes events to the gateway. The gateway forwards these events to peers that take the role of consumers and are subscribed to receive those events. Figure 9.1 shows the software architecture of a gateway and its associated peers.

The gateway and peers use the Acceptor [97] and Connector [97] patterns to establish connections. Once the connections are established, communication takes place between the various service handlers as is shown in Figure 9.1. A service handler in the supplier publishes events (1) to the gateway which are then received by a *Supplier Handler* in the gateway. There is a unique *Supplier Handler* corresponding to each supplier. This *Supplier Handler* forwards the event to a set of *Consumer Handler*s corresponding to consumers that are subscribed to events from the supplier that supplied the event. The event forwarding is based on a routing table that keeps track of the current subscriptions. The *Consumer Handler*s then forward the event to the corresponding consumers. The gateway and its peers use a Reactor [91, 90, 97] to listen to incoming messages from multiple socket endpoints. The Acceptor, Connector and the service handlers are registered with the reactor.

For our purposes here, we made a slight extension to the functionality of the basic gateway. Before forwarding an event to a consumer, the gateway performs some value-added service specific to that consumer. This is quite common in a real-world gateway, for example, a stock quote supplied by a quote supplier is broadcast to different subscribers and based on the subscription level of the quote subscriber, the gateway may collect and then forward more information on the stock like its performance history.

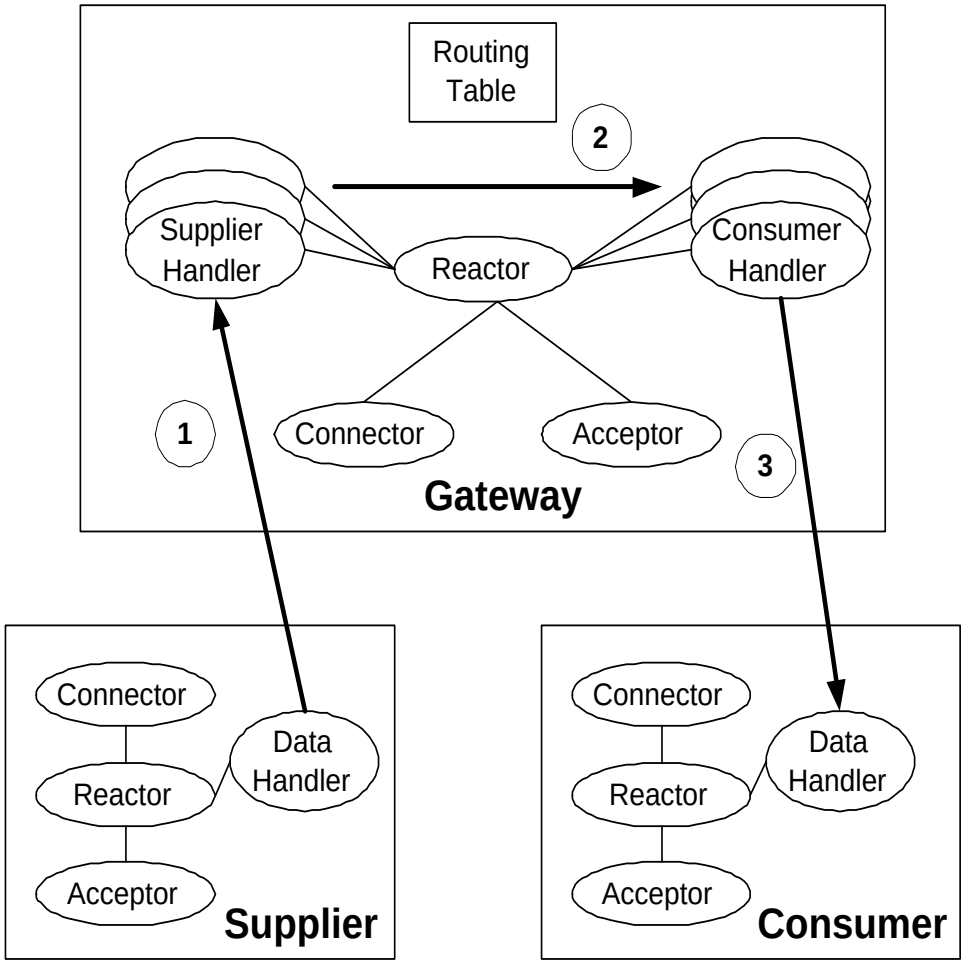


Figure 9.1: Software Architecture of a Gateway

We developed two variations of the Gateway with the same basic functionality of event propagation from suppliers to consumers, with the suppliers being consumer agnostic. The two variations show how application constraints (*e.g.,* for real-time predictability, reliability) drive the middleware configuration which in turn affects the timing and liveness properties of the system. The two variations are (1) a Gateway used in the context of an application with real-time requirements; and (2) a Gateway with reliability requirements used in the context of an application with a control-push-data pull model.

In (1) the requirement is that deadlines for event delivery should be met and in (2) the requirement is that the gateway should send an acknowledgment to a supplier for each event that was published by the supplier. This acknowledgment should happen after all the subscribed consumers have received the published event. In (2) the application uses a control-push-data-pull model in which the supplier publishes a "data available" event to the gateway, the gateway forwards it to the subscribed consumers. The consumers then make a remote call to the supplier to get the actual data. The control-push-data-pull model is widely used, *e.g.,* in DRE systems in the avionics mission computing domain [32, 30, 43]. Apart from illustrating the interference effects arising from the sharing of resources at the middleware layer, these two scenarios are representative of how middleware needs to be designed/configured appropriately to meet different QoS requirements of applications. Our models help the middleware designer/configurator to use a formal basis upon which to choose the appropriate set of middleware configurations.

9.2 Real-time Gateway

In this section, we consider the usage of the Gateway in the context of a real-time application. We consider a scenario using the gateway example, where there is a periodic supply of events from two suppliers. Table 9.1 shows the suppliers and consumers and the periods for the suppliers. Events from S1 are forwarded to consumers C1 and C2 and events from S2 are forwarded to C2 and C3. Note that C2 receives events from both S1 and S2. The deadlines for the arrival of these events at the consumers

Table 9.1: **Periodic Tasks in the Gateway Example**

Supplier	Period	Value add service	Relative deadline	Consumer
S1	100ms	20ms	100ms	C1,C2
S2	50ms	10ms	50ms	C2,C3

is the same as the period of the supplier that supplies the events. The value-add computation for the events supplied by S1 takes 20ms and that for S2 takes 10ms.

9.2.1 High Level Modeling Using RMA

During high level modeling, we try to determine whether the application described by Table 9.1 is schedulable under the given parameters. Typically for a periodic system like this, Rate Monotonic Analysis (RMA) [64, 58, 15, 62, 99] is used to determine whether sharing the same CPU among tasks leads to any timing violations. RMA uses a computation model which uses a "task" as its basis. A "task" is an abstraction of a computation that requires the CPU, and each task has a period and deadline associated with it. We now model the real-time application using the RMA computation model. Assuming that there is a constant propagation delay from the suppliers to the gateway, the events arrive at the gateway at regular intervals. Under the RMA model, the gateway can thus be considered to be a periodic system with 2 periodic tasks. We now do a schedulability analysis to determine whether the CPU resource on the gateway endsystem can be scheduled between the two information flows.

According to RMA, the feasible utilization bound [64, 58, 15] for tasks with harmonic periods is 100%. This means that if the combined utitization of all the tasks taken together is less than 100%, then the system is guaranteed to be schedulable. In the above example, we have the combined utilization = 20*2/100 + 10*2/50 = 0.80 (80%). Note that we multiply the 20ms and 10ms value-added service execution times by 2 to account for the execution time for each consumer. Since the total utilization is well below the utilization bound, the system is guaranteed to be schedulable if the higher frequency task is given a higher priority. Ignoring variation in propagation

delays, this means that if the consumers should also receive events at a periodic rate and there should be no deadline misses.

9.2.2 Design and Implementation

Having done a high-level analysis, we now proceed to implementing the gateway using the architecture in Figure 9.1. There are several design choices [97] available. The purpose here is to show that different design choices impact the application in different ways and to show how our models help the designer in identifying design choices that are free from hazards like deadline misses. If these design choices are not taken into consideration during modeling, then some of the assumptions made during high-level modeling may be violated. Note that in the RMA analysis, we only considered the sharing of resources at the hardware level and did not consider the sharing of resources that could take place at the middleware level. This lack of sufficient detail in the high-level model in turn may lead to a violation of system timing properties during system execution, unless we use a sufficiently detailed model to capture the effects of various design choices thereby guiding the designer to make the appropriate choice.

We modeled the gateway example using the models described in Chapter 5. We modeled the connection establishment phase using acceptor and connector models, which are specialized event handlers used in the gateway example to create service handlers and then populate them with the SAP handles that represent a connection. Once connections have been established, the two suppliers start publishing data on a periodic basis. We kept a log of when each supplier sent a message and when that message reached the consumer. If the elapsed time exceeds the deadline for that message, then there is a deadline miss. As soon as there was a deadline miss we stop the state space exploration using an IF cut observer.

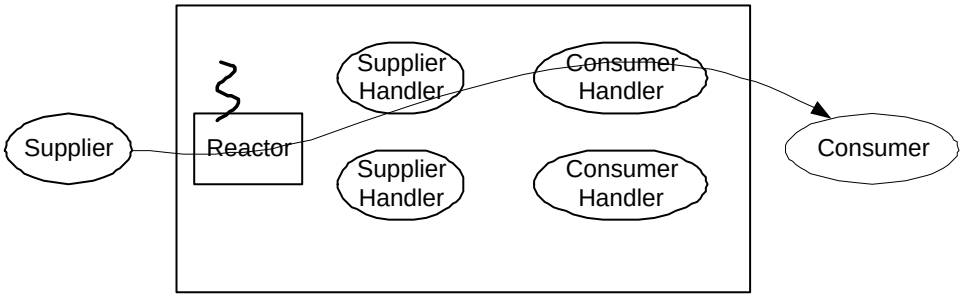
9.2.3 Evaluating Design Alternatives

Design 1: Single Reactor Thread.

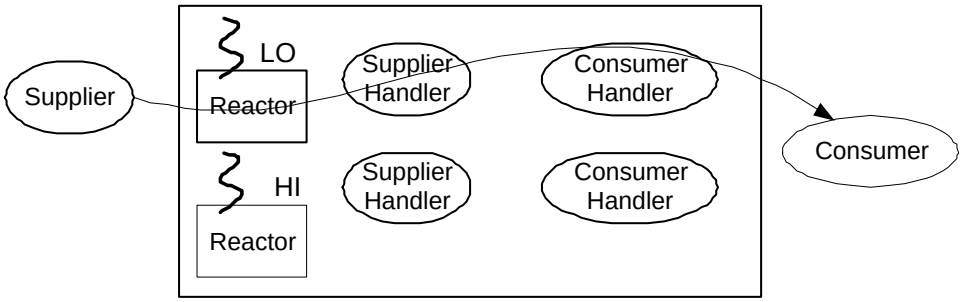
Under this design choice, shown in Figure 9.2(a), there is an I/O thread that waits on socket events using a reactor. When an event arrives from a supplier, the reactor makes an upcall to the appropriate supplier handler which then forwards the event to the appropriate consumer handlers. The consumer handlers send these events to the consumers in the context of the I/O thread itself. Note that the value-added service is also done in the context of the I/O thread.

A simulation using a single thread in the gateway indicated deadline misses. Figure 9.3(a) shows a timeline trace generated from post-processing the IF output trace showing the sequence of events that led to a deadline miss. At the start, the two suppliers respectively send messages, that are received by the gateway reactor which makes an upcall to the supplier handler corresponding to supplier S1. The supplier handler forwards the event to the consumer handlers corresponding to C1 and C2. Since there is only one thread, the forwarding of events is done sequentially. In this case, the supplier handler for S1 forwards the message to the consumer handler corresponding to C1. The consumer handler for C1 does some value-added service for 20 time units and then forwards the event to the consumer C1 which receives the message sent by supplier S1 at time 20. The S1 supplier handler in the gateway then forwards the message from supplier S1 to the gateway consumer handler corresponding to consumer C2. The value-added service is performed for this consumer and then the event is forwarded to C2, which receives the message at time=40. The gateway reactor now makes an upcall to the S2 supplier handler and the sequence is repeated for the message from S2 which is received by C2 at time=50. Note that at time 50, supplier S2 fires again sending a message. However the gateway is still in the middle of processing the first message sent by S2. Consumer C3 receives the first message sent by S2 at time=60. Since the deadline for receiving this message is 50, a deadline miss is detected.

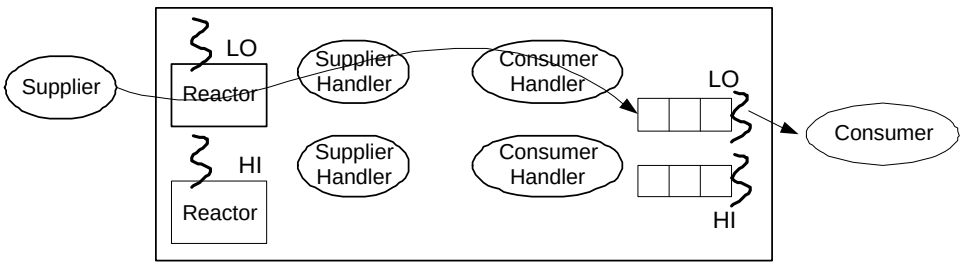
The model execution trace shows that the deadline miss occurred because of a priority inversion that occurred at the reactor in the gateway. The priority inversion occurred



(a) Gateway Implementation Using a Single Thread

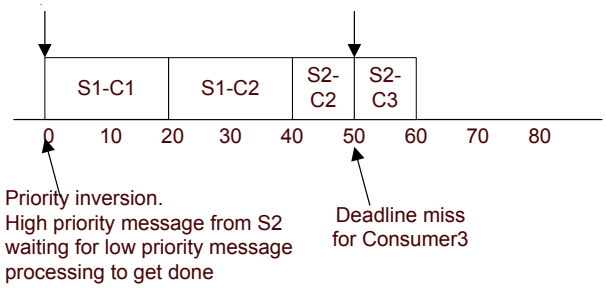


(b) Gateway Implementation Using Reactor Priority Lanes

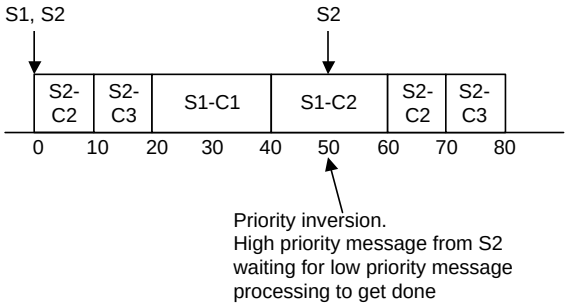


(c) Gateway Implementation Using Dispatch Lanes

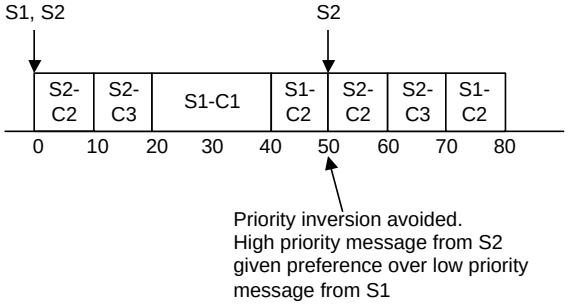
Figure 9.2: Gateway design alternatives



(a) Timeline With Single Reactor Single Thread



(b) Timeline With Reactor Priority Lanes



(c) Timeline With Reactor Priority Lanes and Dispatch Lanes

Figure 9.3: Timelines from Model Execution

because of the sequential nature of the reactor upcalls. Message from S1 was processed first and then the message from S2 was processed. This resulted in a blocking delay for the message from S2. The blocking delay is the time it took for the value-added processing for the message from S1, which in the above example was 40 time units (20 time units each for C1 and C2). This trace thus shows that the enforcement of the high-level RMS model is not achieved using this design approach.

Design 2: Reactor Priority Lanes

To eliminate the priority inversion due to blocking at the reactor in Design 1, we now use separate reactor/thread pair to handle I/O events corresponding to the two suppliers. This design [80, 77] has been used in avoiding priority inversions in real-time ORBs like TAO. Under this design, shown in Figure 9.2(b), there is an I/O thread and reactor per rate group. Each I/O thread waits on socket events using a different reactor. When an event arrives from a supplier, the appropriate reactor (HI or LO) makes an upcall to the appropriate supplier handler which then forwards the event to the appropriate consumer handlers. The consumer handlers send these events to the consumers in the context of the I/O thread itself. Note that the value-added service is also done in the context of the I/O thread. To protect the same event handler (for example the consumer handler for C2) from concurrent upcalls from different reactor threads, access to the event handlers is synchronized.

We again modeled the gateway, using this design choice. Even though there were no deadline misses, the model execution trace showed a priority inversion because of blocking at the synchronized event handler corresponding to consumer C2. Figure 9.3(b) shows a timeline trace generated from post-processing the IF output trace showing the sequence of events that leads to a priority inversion. At the start, the two suppliers respectively send messages, that are received by the corresponding reactors which make upcalls to the supplier handlers. Note that there is no priority inversion at the I/O layer because of the priority isolation achieved by separation of I/O handling for the events from the two suppliers. However the model execution trace shows that a priority inversion occurs at the synchronized consumer handler corresponding to consumer C2. This occurs because the value-added service corresponding to the event from S1 being forwarded to C2 is done by the synchronized consumer handler

for C2. This results in the second event from S2 (released at time = 50) waiting for access to this consumer handler which is being used for the event processing for the event from S1 to C2. Even though this priority inversion does not result in a deadline miss in this example, it may lead to one under other conditions and thus should be eliminated.

Design 3: Reactor Lanes and Dispatch Lanes.

To eliminate the priority inversion due to blocking at the synchronized event handler in Design 2, we now use a separate worker thread to forward the events to the consumer. Under this design, a consumer handler hands over an event to an active object [61, 97], which has its own thread of execution. The events are forwarded to the consumer under the context of the active object thread. The synchronization at the event handler in Design 2 is still there, but the value added service itself is not done within the event handler and is instead done by the active object thread.

To achieve priority isolation for the event dispatching by the active object threads, we use simplified Kokyu [34, 63, 33, 18, 31, 30] based priority lanes for dispatching. In the above example, there are two tasks each with a different period. The number of lanes are based on the number of rate groups - 2 lanes in the above example, since we have two rate groups (100ms and 50ms).

Our model execution traces indicated no deadline misses or priority inversion as is shown in Figure 9.3(c). Note that in this case, we stop the trace at 100 time units. This trace is sufficient to demonstrate absence of these hazards since the hyper-period for the above tasks is 100ms after which the same sequence of events repeat. In this model, we assigned priorities to the lane threads according to RMS. The lane corresponding to the rate group for 100ms was given a lower priority than that for 50ms. As a result, the S1-C2 event processing by the low priority active object thread is preempted (at time=50 units) by the S2-C2 event processing by the high priority active object thread.

Feedback to higher level model. We now take the blocking factor information obtained from the analysis in Section 9.2.3 and use it to refine the original RMA analysis. Since we have blocking delays, we use RMA analysis with blocking factors [15]. For two tasks (T1,C1,B1) and (T2,C2,B2), where T is the period, C is the computation time and B is the blocking delay and T1<T2, the tasks are guaranteed to be schedulable if C1/T1 + B1/T1 <= 1 and C1/T1 + C2/T2 + B2/T2 <= 1 (for harmonic periods). In our example with C1=20, T1=50, B1=40 and C2=40, T2=100, B2=20, these equations are not satisfied and hence schedulability using RMS is not guaranteed. This analysis shows how the information obtained from our middleware models can be fed back to higher level models, thus enabling more faithful analysis.

9.2.4 Empirical Validation

Figure 9.4 shows that our models reflect the actual implementation closely. We implemented the three design alternatives that we discussed earlier and populated our models with the execution timing information from the actual runs. The priority inversion and deadline misses predicted by the models showed up in the actual runs also thus demonstrating the validity of our models. Moreover, we populated the models with execution times from the actual runs and then generated timeline traces from the resulting model execution. The timelines from the model execution trace resembled the timelines from actual execution trace very closely, demonstrating the fidelity of our models.

Figure 9.4 shows the timelines generated from actual and model execution. Figure 9.4(a) shows that in actual execution also, there is a scenario where a priority inversion occurs at time=0, when the single reactor thread dispatches a message from S1, whereas the message from S2 is waiting to be processed. After populating the model with actual execution timing, the model execution resembles the actual execution very closely. Figure 9.4(b) shows that in actual execution also, there is a scenario where a priority inversion occurs at time=50, when the high priority reactor thread is blocked on the shared consumer handler that forwards messages to consumer C2. Even though there is a message from supplier S2 (higher priority) waiting to be processed, the value added service for the message from supplier S1 to consumer C2

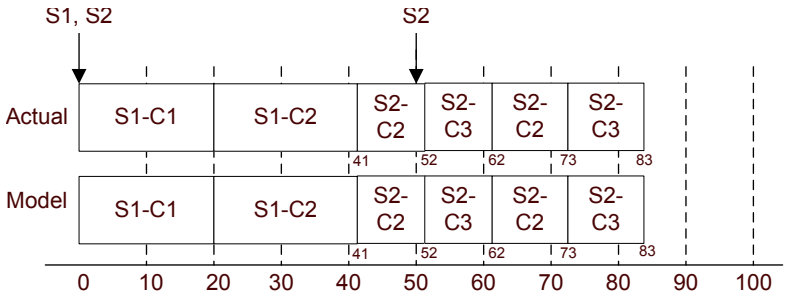
continues till time=69. Figure 9.4(c) shows that these two priority inversions do not occur in actual execution with the addition of the dispatch lanes.

9.3 Reliable Gateway with Control-Push-Data-Pull

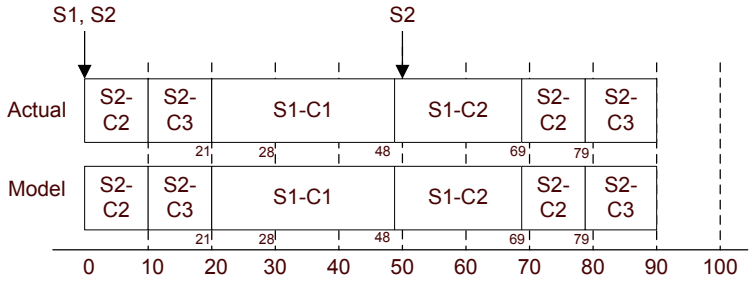
Our discussion so far has concentrated on one form of interference - blocking delays caused by the single threaded reactor in the Gateway or by the synchronized event handler. This form of interference cannot be detected by higher-level computational models such as those offered by RMA (or even DREAM [66]), since these techniques provide computation models that do not allow direct representation of the middleware elements whose sharing causes this interference. In contrast, the blocking delays can be captured by our computational model by virtue of its inclusion of lower-level middleware building blocks. We now illustrate the second form of interference that our models capture - exhaustion of reactor threads - using an application with reliability requirements. This example reemphasizes the fact that such interference can be captured effectively by including lower-level middleware building blocks in our analysis.

In this scenario, the application uses an event propagation model called "control-push-data-pull". In this model, the supplier publishes a "data-available" event to the gateway and the gateway forwards it to the subscribed consumers. The consumers then make a remote call to the supplier to get the actual data. The control-push-data-pull model is widely used in DRE systems. Note that the data availability event (control-push) flows through the gateway whereas the request from the consumer to the supplier (data-pull) takes place outside of the gateway through a remote call.

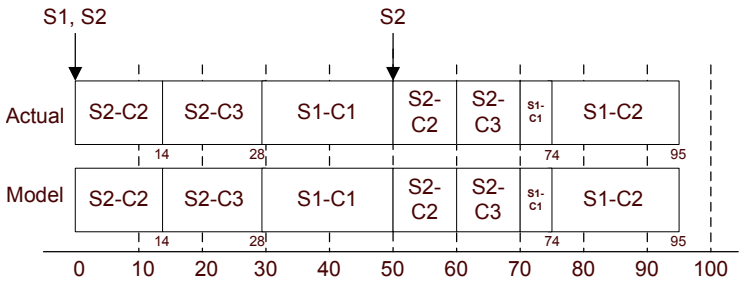
Apart from the control-push-data-pull model, the application also has a reliability requirement - every event that is published by a supplier must be acknowledged by the gateway and every event received by a consumer from the gateway must be acknowledged by that consumer. Once the gateway receives acknowledgments from all the consumers for an event, it sends an acknowledgment back to the supplier. Note that the basic functionality of the gateway is still intact since the suppliers need not keep track of the consumers. However the additional requirements of reliability and



(a) Timeline With Single Reactor Single Thread



(b) Timeline With Reactor Priority Lanes



(c) Timeline With Reactor Priority Lanes and Dispatch Lanes

Figure 9.4: Comparision of Actual and Model Execution Timelines

control-push-data-pull semantics require us to devise new strategies at the middleware level of the gateway implementation. We discuss one such strategy, which is the reply wait strategy which we described in Chapters 1 and 6. For the purposes of the discussion and subsequent modeling and experimentation, we focus on the use of this strategy in the middleware layer and at the suppliers.

To wait for an acknowledgment from the gateway after publishing an event, a supplier could use either the WaitOnConnection or the WaitOnReactor strategy. We now analyze the impact of these two choices on the liveness of the system. We have illustrated in Chapter 6 that the WaitOnConnection strategy can result in deadlock when there is a circular call-chain and that the WaitOnReactor strategy prevents this deadlock by means of a recursive call to the reactor event loop. We now illustrate this in the context of the Gateway example and hence we do not show the timelines and instead show only the relevant events without any timestamps.

We enhanced both our low-level models and our implementation of the Gateway example in ACE by making suitable modifications to accommodate the new requirements of reliability. In the following discussion, we consider a single-threaded reactor implementation of the gateway, where the reactor thread is responsible for demultiplexing event dispatches among connections from suppliers and also for forwarding the events to consumers. We also assume the suppliers and consumers have single-threaded reactors.

9.3.1 Reply Wait Using WaitOnConnection

Using the WaitOnConnection reply wait strategy resulted in a deadlock. An informal analysis shows that this was caused by a nested call-chain, where the supplier waits for an acknowledgment from the gateway which in turn waits for an acknowledgment from the consumer, which in turn waits for data from the supplier. Since there is only one thread in the supplier, this results in a deadlock. We verified the above hypothesis by running an exhaustive simulation of our executable model and the model checker shows the sequence of events that led to the deadlocked state.

Figure 9.5 shows the interaction trace generated after post-processing the output trace from the IF model checker. For clarity, we again do not show the system initialization phase where connections between gateway and suppliers/consumers are established, since it does not affect the outcome of verification. The trace shows that the two suppliers published their respective events (lines 3,5) which are then transported (lines 4,6) by the IPCC automata to the appropriate SAP buffers on which the reactor in the gateway was listening. The gateway reactor unblocked (line 7) and made the upcall (lines 8-11) to the appropriate event handler. The event handler corresponding to S1 forwarded (lines 12-13) the event to the consumer handler for C1 which then performed (lines 14-22) the value-added service and forwarded (line 23) the event to the consumer C1. The reactor in the consumer unblocked (line 25-26) and made an upcall (27-28) and finally the consumer received (line 29) the event. The consumer then sent (line 30) a request to the supplier and after this no transitions were enabled and time advanced to a large preset number (10000) indicating a deadlock.

```
2: {Main}0 ---INIT_MODE_DONE()---> {nil}0
3: {Supplier_Data_Handler}0 ---SUPP_SEND_EVENT(1,1)---> {nil}0
4: Unidir_IPC_17_16 : TRACE_SAP_Buffer_Transfer(17,16,10)
5: {Supplier_Data_Handler}1 ---SUPP_SEND_EVENT(2,1)---> {nil}0
6: Unidir_IPC_20_19 : TRACE_SAP_Buffer_Transfer(20,19,10)
7: Reactor1_SRHE0 : TRACE_Reactor_IO_Wait_Done({16,19,},{})
8: Reactor1_SRHE0 ---SELECT_REACTOR_AFTER_SELECT({16,19,},{})---> {nil}0
9: Reactor1_SRHE0 ---SELECT_REACTOR_BEFORE_UPCALL()---> {nil}0
10: Reactor1_SRHE0 ---handle_input(16,7)---> {Supplier_Connxn_Handler}0
11: {Supplier_Connxn_Handler}0 ---GW_SUPP_HNDLR_HANDLE_INPUT(1,1)---> {nil}0
12: {Supplier_Connxn_Handler}0 ---forward_event({size=10,supp_id=1,event_num=1,cons_id=1})
    ---> {Consumer_Connxn_Handler}0
13: {Supplier_Connxn_Handler}0 ---GW_SUPP_HNDLR_FWD_EVT_TO_CONS_HNDLR(1,1,1)---> {nil}0
14: {Consumer_Connxn_Handler}0 ---GW_CONS_HNDLR_VALUE_ADD_SVC_SLICE_BEGIN(1,1,1)---> {nil}0
15: Time advanced by 10 units. Global time is 11
16: {Consumer_Connxn_Handler}0 ---EXEC_SLICE_SO_FAR(10)---> {nil}0
17: {Consumer_Connxn_Handler}0 ---GW_CONS_HNDLR_VALUE_ADD_SVC_SLICE_DONE(1,1,1)---> {nil}0
18: {Consumer_Connxn_Handler}0 ---GW_CONS_HNDLR_VALUE_ADD_SVC_SLICE_BEGIN(1,1,1)---> {nil}0
19: Time advanced by 10 units. Global time is 21
20: {Consumer_Connxn_Handler}0 ---EXEC_SLICE_SO_FAR(20)---> {nil}0
21: {Consumer_Connxn_Handler}0 ---GW_CONS_HNDLR_VALUE_ADD_SVC_SLICE_DONE(1,1,1)---> {nil}0
22: {Consumer_Connxn_Handler}0 ---GW_CONS_HNDLR_VALUE_ADD_SVC_END(1,1,1)---> {nil}0
23: {Consumer_Connxn_Handler}0 ---GW_CONS_HNDLR_FWD_EVT_TO_CONS(1,1,1)---> {nil}0
24: Unidir_IPC_22_23 : TRACE_SAP_Buffer_Transfer(22,23,10)
25: Reactor3_SRHE0 : TRACE_Reactor_IO_Wait_Done({23,},{})
26: Reactor3_SRHE0 ---SELECT_REACTOR_AFTER_SELECT({23,},{})---> {nil}0
27: Reactor3_SRHE0 ---SELECT_REACTOR_BEFORE_UPCALL()---> {nil}0
28: Reactor3_SRHE0 ---handle_input(23,3)---> {Consumer_Data_Handler}0
29: {Consumer_Data_Handler}0 ---CONS_GOT_EVENT(1,1,1)---> {nil}0
30: Unidir_IPC_2_1 : TRACE_SAP_Buffer_Transfer(2,1,1)
31: {Idle_Catcher}0 ---IDLE_CATCHER_RUNS()---> {nil}0
32: Time advanced by 9979 units. Global time is 10000
```

Figure 9.5: Model Execution Trace with WaitOnConnection

A very short extract from the state space exploration output from IF is shown in Figure 9.6, which shows the final states of the relevant automata when the deadlock occurred. Note that the supplier S1 ({Supplier_Data_Handler}0) was waiting for the acknowledgment from the gateway, consumer C1 ({Consumer_Data_Handler}0) was waiting for reply from the supplier S1 and the gateway ({Consumer_Connxn_Handler}0) was waiting for an acknowledgment from the consumer C1.

```
{Supplier_Data_Handler}0                {}
  @wait_for_ack_on_conn          {....state variables.....}
{Consumer_Data_Handler}0                {}
  @wait_for_reply                {....state variables.....}
{Consumer_Connxn_Handler}0              {}
  @wait_for_ack_from_consumer    {....state variables.....}
```

Figure 9.6: Relevant States at Deadlock with WaitOnConnection

We also verified that the deadlock existed in the actual implementation by implementing the WaitOnConnection reply wait strategy. The post-processed DSUI trace is shown in Figure 9.7. The trace shows that once consumer C1 received the first message there was no further progress, indicating a deadlock.

```
1: SUPP_SEND_EVENT(S2,M1)
2: SUPP_SEND_EVENT(S1,M1)
3: GW_SUPP_HNDLR_HANDLE_INPUT(S1,M1)
4: GW_SUPP_HNDLR_FWD_EVT_TO_CONS_HNDLR(S1,M1,C1)
5: GW_CONS_HNDLR_VALUE_ADD_SVC_BEGIN(S1,M1,C1)
6: GW_CONS_HNDLR_VALUE_ADD_SVC_END(S1,M1,C1)
7: GW_CONS_HNDLR_FWD_EVT_TO_CONS(S1,M1,C1)
8: CONS_GOT_EVENT(S1,M1,C1)
```

Figure 9.7: Actual Execution Trace with WaitOnConnection

9.3.2 Reply Wait Using WaitOnReactor

Using the WaitOnReactor reply wait strategy eliminates the deadlock that arose due to loops in the call-chain. An informal analysis shows that the supplier while waiting for an acknowledgment from the gateway waits on its reactor rather than on a specific connection. This enables the remote request from a consumer to be processed even while the supplier is waiting for the acknowledgment from the gateway. We verified

the above hypothesis by running an exhaustive simulation of our executable model and the exhaustive simulation never produced a deadlock.

Figure 9.8 shows extracts from the interaction trace with the WaitOnReactor strategy. For clarity reasons, we again do not show the entire trace. Instead we show the complete sequence for one published event and all other events follow a similar sequence. The trace is similar to the one for the WaitOnConnection strategy until the consumer C1 got (line 39) an event and sent a request to the supplier S1 and waited for a reply. The sent request reached (line 40) the supplier which was waiting on its reactor. The reactor unblocked (line 41) and made an upcall to handle (lines 42-44) the request from the consumer. The supplier then sent a reply which is then carried (line 47) to the consumer, which then received (line 53) the reply. The consumer then sent an acknowledgment to the gateway which is received (line 56) by the gateway. The sequence of events - (1) forwarding by the gateway to the consumer, (2) the consumer sending a request to the supplier, (3) the consumer receiving the reply and then (4) sending an acknowledgment to the gateway - is repeated (lines 60-98) for consumer C2. After this the gateway sent an acknowledgment to the supplier S1 (lines 102-110). The above sequence of interactions is then repeated for the message from supplier S2 (not shown). This trace shows that the WaitOnReactor strategy eliminated the deadlock arising from the loop in the call-chain.

We also verified that the WaitOnReactor strategy eliminated deadlock in the case of actual execution. A portion of the post-processed DSUI trace from the actual execution is shown in Figure 9.9. This trace shows the complete sequence of events until supplier S1 receives an acknowledgment from the gateway. This same sequence is repeated for supplier S2, which we have not shown here. The trace shows that supplier S1 received (line 9) the request sent to it by C1 since S1 was waiting on the reactor instead of a connection. There is no deadlock in this case as opposed to the deadlock seen when using the WaitOnConnection strategy to wait for the reply. Because of the WaitOnReactor strategy, each message sent by the suppliers is acknowledged (line 20) by the gateway without causing a deadlock.

```
.........................
39: {Consumer_Data_Handler}0 ---CONS_GOT_EVENT(1,1,1)---> {nil}0
40: Unidir_IPC_2_1 : TRACE_SAP_Buffer_Transfer(2,1,1)
41: Reactor2_SRHE1 : TRACE_Reactor_IO_Wait_Done({1,},{})
42: Reactor2_SRHE1 ---SELECT_REACTOR_AFTER_SELECT({1,},{})---> {nil}0
43: Reactor2_SRHE1 ---SELECT_REACTOR_BEFORE_UPCALL()---> {nil}0
44: Reactor2_SRHE1 ---handle_input(1,1)---> {Consumer_Request_Handler}0
45: {Consumer_Request_Handler}0 ---SUPP_RECVD_REQ_FROM_CONS()---> {nil}0
46: {Consumer_Request_Handler}0 ---handle_input_return(1)---> Reactor2_SRHE1
47: Unidir_IPC_1_2 : TRACE_SAP_Buffer_Transfer(1,2,1)
48: Reactor2_SRHE1 ---SELECT_REACTOR_AFTER_UPCALL()---> {nil}0
49: Reactor2_SRHE1 ---handle_events_return()---> {Supplier_Data_Handler}0
50: {Supplier_Data_Handler}0 ---handle_events(1)---> Reactor2
51: Reactor2 forks Reactor2_SRHE1
52: Reactor2_SRHE1 ---SELECT_REACTOR_BEFORE_SELECT()---> {nil}0
53: {Consumer_Data_Handler}0 ---CONS_RECVD_DATA_FROM_SUPP()---> {nil}0
54: {Consumer_Data_Handler}0 ---handle_input_return(0)---> Reactor3_SRHE0
55: Unidir_IPC_23_22 : TRACE_SAP_Buffer_Transfer(23,22,1)
56: {Consumer_Connxn_Handler}0 ---GW_RECVD_ACK_FROM_CONS()---> {nil}0
.........................
60: {Supplier_Connxn_Handler}0 ---GW_SUPP_HNDLR_FWD_EVT_TO_CONS_HNDLR(1,1,2)---> {nil}0
.........................
81: {Consumer_Data_Handler}1 ---CONS_GOT_EVENT(1,1,2)---> {nil}0
.........................
87: {Consumer_Request_Handler}0 ---SUPP_RECVD_REQ_FROM_CONS()---> {nil}0
.........................
95: {Consumer_Data_Handler}1 ---CONS_RECVD_DATA_FROM_SUPP()---> {nil}0
.........................
98: {Consumer_Connxn_Handler}1 ---GW_RECVD_ACK_FROM_CONS()---> {nil}0
.........................
102: Unidir_IPC_16_17 : TRACE_SAP_Buffer_Transfer(16,17,1)
103: Reactor2_SRHE1 : TRACE_Reactor_IO_Wait_Done({17,},{})
104: Reactor2_SRHE1 ---SELECT_REACTOR_AFTER_SELECT({17,},{})---> {nil}0
105: Reactor2_SRHE1 ---SELECT_REACTOR_BEFORE_UPCALL()---> {nil}0
106: Reactor2_SRHE1 ---handle_input(17,1)---> {ACK_Handler}0
107: {ACK_Handler}0 ---handle_input_return(1)---> Reactor2_SRHE1
108: Reactor2_SRHE1 ---SELECT_REACTOR_AFTER_UPCALL()---> {nil}0
109: Reactor2_SRHE1 ---handle_events_return()---> {Supplier_Data_Handler}0
110: {Supplier_Data_Handler}0 ---SUPP_RECVD_ACK_FROM_GW()---> {nil}0
```

Figure 9.8: Model Execution Trace with WaitOnReactor

```
1: SUPP_SEND_EVENT(S2,M1)
2: SUPP_SEND_EVENT(S1,M1)
3: GW_SUPP_HNDLR_HANDLE_INPUT(S1,M1)
4: GW_SUPP_HNDLR_FWD_EVT_TO_CONS_HNDLR(S1,M1,C1)
5: GW_CONS_HNDLR_VALUE_ADD_SVC_BEGIN(S1,M1,C1)
6: GW_CONS_HNDLR_VALUE_ADD_SVC_END(S1,M1,C1)
7: GW_CONS_HNDLR_FWD_EVT_TO_CONS(S1,M1,C1)
8: CONS_GOT_EVENT(S1,M1,C1)
9: SUPP_RECVD_REQ_FROM_CONS(S1)
10: CONS_RECVD_DATA_FROM_SUPP(C1)
11: GW_RECVD_ACK_FROM_CONS(C1)
12: GW_SUPP_HNDLR_FWD_EVT_TO_CONS_HNDLR(S1,M1,C2)
13: GW_CONS_HNDLR_VALUE_ADD_SVC_BEGIN(S1,M1,C2)
14: GW_CONS_HNDLR_VALUE_ADD_SVC_END(S1,M1,C2)
15: GW_CONS_HNDLR_FWD_EVT_TO_CONS(S1,M1,C2)
16: CONS_GOT_EVENT(S1,M1,C2)
17: SUPP_RECVD_REQ_FROM_CONS(S1)
18: CONS_RECVD_DATA_FROM_SUPP(C2)
19: GW_RECVD_ACK_FROM_CONS(C2)
20: SUPP_RECVD_ACK_FROM_GW(S1)
```

Figure 9.9: Actual Execution Trace with WaitOnReactor

9.4 Summary

In this chapter, we demonstrated the flexibility and reusability of our models in the context of modeling the two variants of an existing example of an application-level gateway that is distributed in the source tree of the ACE framework. We demonstrated how gaps could exist between high level models and actual implementations. We showed how such gaps can be narrowed by using our middleware level models in analyzing the effects of different strategies at the middleware level.

Chapter 10

Conclusions and Future Work

This dissertation has concentrated on investigating the necessity and feasibility of including formal middleware models to perform more faithful analysis of DRE systems behavior. While current approaches for modeling middleware focus largely on easing the task of assembling, deploying and configuring middleware and middleware-based applications, we have shown that a more formal basis for correct middleware construction and configuration in the context of individual applications is helpful. Our approach, presented in Chapter 3, is designed to address that concern.

A combination of computational models and techniques should be used to analyze DRE system properties. Static analysis is cheaper than model checking, for example, RMA analysis is cheaper and works for a lot of predominantly time-driven systems. However, when we *implement* systems using middleware building blocks there is often accidental complexity involved when making design choices, as we showed in the gateway example in Chapter 9. This accidental complexity can be analyzed by developing models that include the middleware also. Another example we showed was the deadlock avoidance protocol described in Chapter 8. This protocol was proven elsewhere [88, 89] to avoid deadlock and we demonstrated that both using model checking and empirical verification also. The accidental complexity in that example was the blocking factor introduced by the protocol, which was revealed by our models.

The examples presented in Chapters 6, 8 and 9 illustrate a variety of ways in which evaluating timing and liveness properties can be complicated by different combinations of middleware mechanisms. In practice, the range of complicating factors is much larger than even these examples show, which motivates both our development

of reusable mechanism-level models and our composition-based model checking approach for analysis of entire systems. For example, different applications will naturally exhibit (1) different dependency topologies between event handlers; (2) various strategies for concurrency, scheduling, event demultiplexing, and other crucial mechanisms; (3) alternative strategies for handlers relinquishing control during blocking actions, such as WaitOnConnection and WaitOnReactor; and (4) multiple additional on-line protocols, *e.g.,* for deadlock avoidance, real-time admission control, or security authorization. Furthermore, the *constraints* each application places on timing and other properties will alter the criteria by which system timeliness and liveness are evaluated.

Deadlock avoidance protocols guarantee deadlock freedom under certain conditions, and in some cases analysis can be used to determine a number of threads in each reactor that would avoid deadlock without use of a run-time deadlock avoidance protocol. Model checking then can be used to verify whether there are any deadline misses in the system resulting from a variety of blocking factors. The results of our simulations and experiments presented in Chapter 6 motivate the need for detailed modeling of low-level middleware mechanisms, and evaluation of those models through model checking tools. With or without additional protocols, our models can be used for model checking behavior of systems built using the middleware primitives we have modeled. Therefore, the results of our evaluations support our contention that modeling and analysis should be done as an integral part of the system design and engineering process. Significant further work is needed to make this vision a reality in the DRE middleware domain, but the work presented in this dissertation motivates the suitability and viability of that approach.

In Chapters 4 and 5, we identified several engineering challenges associated with modeling middleware and presented solutions addressing those challenges. These techniques can be used as a guide when modeling concurrent object middleware using an appropriate formalism. The state space exploration results shown in Table 6.3 in Chapter 6 showed that caution must be exercised in choosing features offered by the modeling tool. In particular, the choices must be evaluated in the context of the application that is being modeled.

In the research presented in this dissertation, we have focused on creating models for combinations of middleware mechanisms and evaluating them with the model checking tools UPPAAL and IF [12]. Our long term objective is to add further rigor to the model-based approaches to middleware development currently being pursued by the systems research community, and to provide high-fidelity composable models of foundational middleware building blocks to the formal methods community.

10.1 Summary of Contributions

In summary, this dissertation makes the following contributions to research on model-driven middleware, benefiting both the middleware development community and the formal modeling community:

1. A computational model and a modeling architecture based on timed automata for a core subset of the middleware building blocks that are reified in the ACE [51] framework.
2. Concrete engineering challenges and solutions for modeling concurrent object middleware using UPPAAL and IF.
3. Composable and reusable models of commonly used middleware building blocks.
4. A variety of examples that illustrate the composability and flexibility of our executable models.
5. Tracing tools and techniques that help debugging and analysis of models.

10.2 Future Work

The goal of our research is to address the problem of evaluating real-world complex middleware environments, while preserving both rigor in analysis and tractability in applying our approach to real world systems. To meet that goal our future work will focus on developing an ever-expanding set of robust, modular, and composable models

of middleware building blocks, and integrating those models within model-integrated computing tool sets such as those described in Chapter 2. We will also continue our work on formally verified efficient protocols [88], along with the other optimizations described in Chapter 5 to both expand the expressive power and reduce the burden of model checking. Since our models are executable models, they can be used to run guided simulations to verify specific scenarios in cases where exhaustive state space exploration of all possible scenarios is intractable. Future research directions of interest include the following topics.

Trace equivalence. As the complexity of the models increases with respect to concurrency and non-determinism, the number of trace sequences generated by the model checker may grow exponentially. Partitioning the traces into different equivalence classes based on specific equivalence criteria will greatly help to make the exploration of the model output tractable as well as to give significant insights into how to reduce the complexity of the models themselves.

Inclusion of jitter in models. During the case studies that we discussed in this dissertation, we noticed that there is often some degree of temporal jitter during execution of actual implementations whereas during model execution there is no jitter. This is because the model runs under virtual time which is controlled by the model checker. It is possible to include jitter as part of the model thus making the models explore all possible execution paths taking execution jitter also into consideration. Initial results show that inclusion of jitter increases the state space tremendously. We want to investigate this further to analyze its effects, and to understand the trade-offs between fidelity and tractability that these initial results imply.

Use of middleware models in adaptive system reconfiguration. Mission-critical computing systems pose numerous research challenges including but not restricted to satisfaction of functional as well as multiple QoS requirements, resilience to hardware and software failures, and multi-level reconfiguration of the system to address different requirements that could change dynamically. With the increasing use of middleware in DRE systems, one of the key challenges is how to reconfigure

the middleware services adaptively but safely. A variety of formal techniques are of potential use to verify that the system conforms to expected behavior in the face of dynamically changing environment. A promising area of research would be to investigate the usage of executable middleware models in formally reasoning about the impact of dynamically and/or statically reconfiguring the middleware services, on the functionality and QoS of the system.

Modeling of OS primitives. Though this is a different domain than the middleware domain that was the focus of this dissertation, many of the results and modeling techniques from this research appear to be applicable to modeling the operating system domain as well. Dr. Douglas Niehaus of the University of Kansas, is developing composable and executable formal models of operating systems primitives. There is further scope for research in the applicability of group scheduling [5] techniques in model checking to reduce the state space.

Tools support. Automatic model transformation from higher level models (like the component model in ESML described in Chapter 2) to our computation model, using *e.g.,* graph transformations and from our computation model to timed automata again using further model transformation. The idea is that the high level model could be annotated with the deployment details, *e.g.,* threads, reactors, reply wait strategies, select/tp-reactors. The annotations could be done in one stage and then an analysis stage could traverse the model and collect middleware or platform level details and convert them to a middleware computational model.

Low-level component models. Another promising direction of future work would be to investigate imperative component models like Koala, which connect components together with "requires" and "provides" ports. These connections represent method bindings. The interaction between the components is primarily sequential. It would be potentially useful to investigate such component models to see whether they provide (or could be extended to provide) sufficient facilities to model the middleware elements that we dealt with in this dissertation. We believe that our work here would be able to provide the necessary behavioral semantics to the "components" in such

static component models, just as CADENA complements CCM, which is mainly a static component model, by enabling the user to provide additional behavioral descriptions of components.

Generative programming. Another area of research is whether light-weight component models can be used to assemble applications using a substrate like ACE. The strong supporting factor for this is that there are classes in ACE that are re-used across different applications and these classes interact and "connect" in similar manner. There are already pattern languages that catalog such interactions. Depending on the definition of a component, we may or may not be able to classify such stitching together of lower level components as a component model, but the same ideas of declarative assembly of components that are found in higher component models could possibly be realized for the assembly of lower-level implementation framework components using generative programming techniques like the C++ static template meta-programming. This technique could be potentially applicable and relevant to application-driven customization of middleware (like nORB or MicroQoSCORBA or UBI-Core) which is another active area of research.

References

[1] Labelled Transition System Analyzer. `http://www.doc.ic.ac.uk/~jnm/book/ltsa/LTSA.html`.

[2] Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java, Vol I. http://ptolemy.eecs.berkeley.edu/papers/05/ptIIdesign1-intro/ptIIdesign1-intro.pdf, 2005.

[3] 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.

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

[5] Tejasvi Aswathanarayana, Venkita Subramonian, Douglas Niehaus, and Christopher Gill. Design and performance of configurable endsystem scheduling mechanisms. In *Proceedings of 11th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[6] Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Jeff Parsons, Aniruddha Gokhale, and Douglas C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 190–199, San Francisco, CA, March 2005. IEEE.

[7] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A tutorial on uppaal. In *SFM*, pages 200–236, 2004.

[8] Greg Bollella, James Gosling, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-time Specification for Java*. Addison-Wesley, 2000.

[9] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling Urgency in Timed Systems. In *COMPOS*, pages 103–129. Springer-Verlag LNCS 1536, 1997.

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

[11] M. Bozga, S. Graf, Il. Ober, and L. Mounier. IF-2.0: A validation environment for Component-Based Real-time Systems. In *Proceedings of CAV'02*. Springer-Verlag LNCS 2404, 2002.

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

[13] B. Buchanan, D. Niehaus, D. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House. The data stream kernel interface. Technical Report ITTC-FY98-TR11510-04, Information and Telecommunication Technology Center, University of Kansas, 1998.

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture—A System of Patterns*. Wiley & Sons, New York, 1996.

[15] Giorgio C. Buttazzo. *Hard Real-time Computing Systems*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[16] G. J. Chaitin. Register allocation and spilling via graph coloring. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101. ACM Press, 1982.

[17] Kai Chen, Janos Sztipanovits, and Sherif Abdelwahed. A semantic unit for timed automata based modeling languages. In *Proceedings of 12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[18] Christopher D. Gill et al. Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-time Systems. In *Proceedings of the 1st IEEE International Workshop on Real-time Mission-Critical Systems: Grand Challenge Problems*, November 1999.

[19] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.

[20] Alberto Coen-Porisini, Matteo Pradella, Matteo Rossi, and Dino Mandrioli. A formal approach for designing corba-based applications. *ACM Trans. Softw. Eng. Methodol.*, 12(2):107–151, 2003.

[21] G. Coulson and S. Baichoo. Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment. *ACM Distributed Computing Journal*, 14(2), April 2001.

[22] Dionisio de Niz and Raj Rajkumar. Time weaver: a software-through-models framework for embedded real-time systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 133–143, New York, NY, USA, 2003. ACM Press.

[23] William Deng, Matthew B. Dwyer, John Hatcliff, Georg Jung, Robby, and Gurdip Singh. Model-checking Middleware-based Event-driven Real-time Embedded Software. Department of Computer Science, Technical Report SAnToS-TR2003-2, Department of Computing and Information Sciences, Kansas State University, 2003.

[24] Mayur Deshpande, Douglas C. Schmidt, Carlos O'Ryan, and Darrell Brunsch. Design and Performance of Asynchronous Method Handling for CORBA. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications*, Irvine, CA, October/November 2002. OMG.

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

[26] Gregory Duval. Specification and verification of an object request broker. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 43–52, Washington, DC, USA, 1998. IEEE Computer Society.

[27] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby. Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework. In *CAV*, pages 148–152, 2005.

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

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[30] Christopher Gill, Douglas C. Schmidt, and Ron Cytron. Multi-Paradigm Scheduling for Distributed Real-time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), January 2003.

[31] Christopher D. Gill, Ron Cytron, and Douglas C. Schmidt. Middleware Scheduling Optimization Techniques for Distributed Real-time and Embedded Systems. In *Proceedings of the 7^{th} Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, January 2002. IEEE.

[32] Christopher D. Gill, Jeanna M. Gossett, David Corman, Joseph P. Loyall, Richard E. Schantz, Michael Atighetchi, and Douglas C. Schmidt. Integrated Adaptive QoS Management in Middleware: An Empirical Case Study. *Journal of Real-time Systems*, 24, 2005.

[33] Christopher D. Gill, Fred Kuhns, David L. Levine, Douglas C. Schmidt, Bryan S. Doerr, Richard E. Schantz, and Alia K. Atlas. Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-time Systems. In *Proceedings of the 1st IEEE International Workshop on Real-time Mission-Critical Systems: Grand Challenge Problems*, November 1999.

[34] Christopher D. Gill, David L. Levine, and Douglas 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*, 20(2), March 2001.

[35] Patrice Godefroid. Model Checking for Programming Languages using Verisoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[36] Aniruddha Gokhale, Krishnakumar Balasubramanian, Jaiganesh Balasubramanian, Arvind S. Krishna, George T. Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model Driven Middleware: A New Paradigm for Deploying and Provisioning Distributed Real-time and Embedded Applications. *The Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2005 (to appear).

[37] Aniruddha Gokhale, Balachandran Natarajan, Douglas C. Schmidt, Andrey Nechypurenko, Jeff Gray, Nanbor Wang, Sandeep Neema, Ted Bapty, and Jeff 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, November 2002. ACM.

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

[39] Jeffrey Gray, Ted Bapty, and Sandeep Neema. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, pages 87–93, October 2001.

[40] Y. Gu, Y. Li, and D. Towsley. On integrating fluid models with packet simulation. In *Proceedings of IEEE Infocom 2004*, 2004.

[41] Zonghua Gu and Kang Shin. Model-Checking of Component-Based Real-time Embedded Software Based on CORBA Event Service. In *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*. IEEE/IFIP, 2005.

[42] David Harel and Eran Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[43] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-time CORBA Event Service. In *Proceedings of OOPSLA '97*, pages 184–199, Atlanta, GA, October 1997. ACM.

[44] John Hatcliff, William Deng, Matthew Dwyer, Georg Jung, and Venkatesh 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.

[45] T. Henzinger, C. Kirsch, R. Majumdar, and S. Matic. Time safety checking for embedded programs. In *Proc. Second International Workshop on Embedded Software (EMSOFT)*, LNCS. Springer Verlag, 2002.

[46] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: predictable, portable real-time code. *SIGPLAN Not.*, 37(5):315–326, 2002.

[47] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104. ACM Press, 2000.

[48] Ron Hitchens. *Java NIO*. O'Reilly, 2002.

[49] Gerald J. Holtzman. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

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

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

[52] Jeff Kramer Jeff Magee. *Concurrency: State Models and Java Programs*. Wiley, 2000.

[53] Moataz Kamel and Stefan Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. In *Int. Journal on Software Tools for Technology Transfer*. Springer–Verlag, 2000.

[54] Gabor Karsai, Sandeep Neema, Arpad Bakay, Akos Ledeczi, Feng Shi, and Aniruddha 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.

[55] N. Kaveh. Model checking distributed objects design. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 793–794, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[56] Raymond Klefstad, Douglas C. Schmidt, and Carlos 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, April 2002.

[57] Raymond Klefstad, Douglas C. Schmidt, and Carlos 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, March 2002. IEEE/IFIP.

[58] Mark H. Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A Practitioner's Handbook for Real-time Analysis: Guide to Rate Monotonic Analysis for Real-time Systems*. Kluwer Academic Publishers, Norwell, Massachusetts, 1993.

[59] Arvind S. Krishna, Emre Turkay, Aniruddha Gokhale, and Douglas C. Schmidt. Model-Driven Techniques for Evaluating the QoS of Middleware Configurations for DRE Systems. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, pages 180–189, San Francisco, CA, March 2005. IEEE.

[60] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In *Proceedings of the* 2^{nd} *Annual Conference on the Pattern Languages of Programs*, pages 1–7, Monticello, Illinois, September 1995.

[61] R. Greg Lavender and Douglas C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In James O. Coplien, John Vlissides, and Norm Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, Massachusetts, 1996.

[62] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10th IEEE Real-time Systems Symposium (RTSS 1989)*, pages 166–171. IEEE Computer Society Press, 1989.

[63] David L. Levine, Christopher D. Gill, and Douglas C. Schmidt. Dynamic Scheduling Strategies for Avionics Mission Computing. In *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, November 1998.

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

[65] Jie Liu, Xiaojun Liu, and Edward A. Lee. Modeling Distributed Hybrid Systems in Ptolemy II. In *Proceedings of the American Control Conference*, June 2001.

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

[67] Gabor Madl, Sherif Abdelwahed, and Douglas C. Schmidt. Verifying distributed real-time properties of embedded systems via graph transformations and model checking. *International Journal of Time-Critical Computing Systems*, 2005.

[68] Manuel Roman. Ubicore: Universally Interoperable Core. `www.ubi-core.com`.

[69] Manuel Roman and Roy H. Campbell and Fabio Kon. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), July 2001.

[70] Sandeep Neema, Ted Bapty, Jeff Gray, and Aniruddha 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, October 2002.

[71] D. Niehaus. Improving Support for Multimedia System Experimentation and Deployment. In *Workshop on Parallel and Distributed Real-time Systems*, San Juan, Puerto Rico, April 1999. Also appears in Springer Lecture Notes in Computer Science 1586, Parallel and Distributed Processing, ISBN 3–540–65831–9, pp 454–465.

[72] Object Management Group. Model Integrated Computing PSIG. http://mic.omg.org.

[73] Object Management Group. *Lightweight CCM RFP*, realtime/02-11-27 edition, November 2002.

[74] Object Management Group. *Real-time CORBA Specification*, 1.1 edition, August 2002.

[75] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, December 2002.

[76] DARPA Information Exploitation Office. Model-Based Integration of Embedded Software (MoBIES). `www.darpa.mil/ixo/mobies.asp`.

[77] Carlos O'Ryan, Douglas C. Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons, Irfan Pyarali, and David Levine. Evaluating Policies and Mechanisms for Supporting Embedded, Real-time Applications with CORBA 3.0. In *Proceedings of the* 6^{th} *IEEE Real-time Technology and Applications Symposium*, Washington DC, May 2000. IEEE.

[78] OSEK Consortium. OSEK/VDX communication specification. http://www.osek-vdx.org, 2004.

[79] Patrice Godefroid. Software model checking: the VeriSoft approach. Technical Report Technical Memorandum ITD-03-44189G, Bell Labs, 2003.

[80] Irfan Pyarali, Carlos O'Ryan, Douglas C. Schmidt, Nanbor Wang, Vishal Kachroo, and Aniruddha Gokhale. Using Principle Patterns to Optimize Real-time ORBs. *IEEE Concurrency Magazine*, 8(1), 2000.

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

[82] John Regehr, Alastair Reid, Kirk Webb, and Jay Lepreau. Composable Execution Environments. `www.cs.utah.edu/flux/papers/cee-flux-tn-02-02/`, 2002.

[83] Alastair Reid and John Regehr. Task/Scheduler Logic: Reasoning about Concurrency in Component-Based Systems Software. `www.www.cs.utah.edu/~regehr/papers/tsl/tsl-pdf.pdf`, 2002.

[84] Ricardo Santos Marques, Fancoise Simonot-Lion. Design-Patterns based development of an automotive middleware. In *Proceedings of the 6th IFAC International Conference on Fieldbus Systems and their Applications (FeT 2005)*, 2005.

[85] Ricardo Santos Marques, Fancoise Simonot-Lion. Guidelines for the development of a communication middleware for automotive applications. In *Proceedings of the 3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3 2005)*, 2005.

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

[87] Manuel Roman, M. Dennis Mickunas, Fabio Kon, and Roy H. Campbell. LegORB and Ubiquitous CORBA. In *Reflective Middleware Workshop*. ACM/IFIP, April 2000.

[88] Cesar Sanchez, Henny B. Sipma, Venkita Subramonian, Christopher Gill, and Zohar Manna. Thread Allocation Protocols for Distributed Real-time and Embedded Systems. In *25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '05)*, oct 2005.

[89] Cesar Sanchez, Henny B. Sipma, Venkita Subramonian, Christopher Gill, and Zohar Manna. On Efficient Distributed Deadlock Avoidance for Real-time and Embedded Systems. In *20^{th} IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, April 2006.

[90] Douglas C. Schmidt. The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing (Part 2 of 2). *C++ Report*, 5(7), September 1993.

[91] Douglas C. Schmidt. The Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing (Part 1 of 2). *C++ Report*, 5(2), February 1993.

[92] Douglas C. Schmidt. Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services. In *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, Aarhus, Denmark, August 1995.

[93] Douglas C. Schmidt. A Family of Design Patterns for Application-level Gateways. *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, 2(1), 1996.

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

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

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

[97] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[98] Douglas C. Schmidt and Steve Vinoski. Introduction to CORBA Messaging. *C++ Report*, 10(10), November/December 1998.

[99] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized Rate Monotonic Scheduling Theory: A Framework for Developing Real-time Systems. *Proceedings of the IEEE*, 82(1), January 1994.

[100] David C. Sharp and Wendy C. Roll. Model-Based Integration of Reusable Component-Based Avionics System. In *Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, May 2003.

[101] James Snell and Ken MacLeod. *Programming Web Applications with SOAP*. O'Reilly, 2001.

[102] John A. Stankovic, Hexin Wang, Marty Humphrey, Ruiquing Zhu, Ramasubramaniam Poornalingam, and Chenyang Lu. VEST: Virginia Embedded Systems Toolkit. In *Proceedings of the IEEE Real-time Embedded Systems Workshop*, London, UK, December 2001. IEEE.

[103] Venkita Subramonian and Christopher Gill. A Generative Programming Framework for Adaptive Middleware. In *Hawaii International Conference on System Sciences, Software Technology Track, Adaptive and Evolvable Software Systems Minitrack, HICSS 2004*, Honolulu, HW, January 2004. HICSS.

[104] Venkita Subramonian, Guoliang Xing, Christopher Gill, and Ron Cytron. The design and performance of special purpose middleware: A sensor networks case study. Technical Report WUCSE-2003-06, Computer Science and Engineering Department, Washington University in St.Louis, 2003.

[105] Venkita Subramonian, Guoliang Xing, Christopher Gill, Chenyang Lu, and Ron Cytron. Middleware specialization for memory-constrained networked embedded systems. In *Proceedings of 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS)*, 2004.

[106] Janos Sztipanovits and Gabor Karsai. Model-Integrated Computing. *IEEE Computer*, 30(4):110–112, April 1997.

[107] TimeSys. TimeWiz. www.timesys.com, 2002.

[108] Emre Turkay, Aniruddha Gokhale, and Bala Natarajan. Addressing the Middleware Configuration Challenges using Model-based Techniques. In *Proceedings of the 42nd Annual Southeast Conference*, Huntsville, AL, April 2004. ACM.

[109] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz. Total Quality of Service Provisioning in Middleware and Applications. *The Journal of Microprocessors and Microsystems*, 27(2):45–54, mar 2003.

[110] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. An Overview of the CORBA Component Model. In George Heineman and Bill Councill, editors, *Component-Based Software Engineering*. Addison-Wesley, Reading, Massachusetts, 2000.

[111] Martin Fowler with Kendall Scott. *UML Distilled—A Brief Guide to the Standard Object Modeling Language, 2nd Edition*. Addison-Wesley, Boston, 2000.

Vita

Venkita Subramonian

Date of Birth May 17, 1970

Place of Birth Trivandrum, India

Degrees B.S. Computer Science, July 1991, University of Kerala, Trivandrum, India
M.S. Computer Science, May 2000, University of Missouri, Rolla, USA

Book Chapters

Venkita Subramonian and Christopher Gill, "Middleware Design and Implementation for Networked Embedded Systems", in *Embedded Systems Handbook* (Richard Zurawski, ed.), CRC Press, Florida, 2005, Chapter 30, pp. 1-17.

Publications

Venkita Subramonian, Gan Deng, Christopher Gill, Jaiganesh Balasubramanian, Liang-Jui Shen, William Otte, Douglas Schmidt, Andy Gokhale, and Nanbor Wang, "The Design and Performance of Component Middleware for QoS-enabled Deployment and Configuration of DRE Systems", *Elsevier Journal of Systems and Software, Special Issue on Component-Based Software Engineering of Trustworthy Embedded Systems*, 2006.

Cesar Sanchez, Henny Sipma, Venkita Subramonian and Christopher Gill, "On Efficient Distributed Deadlock

Avoidance for Real-Time and Embedded Systems", *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium* (IPDPS), April 2006.

Venkita Subramonian and Christopher Gill, "Towards Integrated Model-Driven Verification and Empirical Validation of Reusable Software Frameworks for Automotive Systems", *Proceedings of Automotive Software Workshop*, March 2006.

Cesar Sanchez, Henny Sipma, Venkita Subramonian and Christopher Gill, "Thread Allocation Protocols for Distributed Real-Time and Embedded Systems", *25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems* (FORTE), October 2005.

Tejasvi Aswathanarayana, Venkita Subramonian, Douglas Niehaus and Christopher Gill, "Design and Performance of Configurable Endsystem Scheduling Mechanisms", *11th IEEE Real-Time and Embedded Technology and Applications Symposium* (RTAS), March 2005.

Venkita Subramonian, Liang-Jui Shen, Christopher Gill and Nanbor Wang, "The Design and Performance of Dynamic and Static Configuration Mechanisms in Component Middleware for Distributed Real-Time and Embedded Systems", *25th IEEE International Real-Time Systems Symposium* (RTSS), December 2004.

Nanbor Wang, Christopher Gill, Douglas Schmidt and Venkita Subramonian, "Configuring Real-time Aspects in Component Middleware", *Distributed Objects and Applications* (DOA), Oct 2004.

Venkita Subramonian, Boris Klaydman and Christopher Gill, "Towards Formal Construction of Middleware for

Distributed Real-Time and Embedded Systems", *Proceedings of the Monterey Workshop on Software Engineering Tools: Compatibility and Integration*, Oct 2004.

Venkita Subramonian, Guoliang Xing, Christopher Gill, Chenyang Lu and Ron Cytron, "Middleware Specialization for Memory-Constrained Networked Embedded Systems", *9th IEEE Real-Time and Embedded Technology and Applications Symposium* (RTAS), May 2004.

Xiaorui Wang, Huang-Ming Huang, Venkita Subramonian, Chenyang Lu and Christopher Gill, "CAMRIT: Control-based Adaptive Middleware for Real-time Image Transmission", *9th IEEE Real-Time and Embedded Technology and Applications Symposium* (RTAS), May 2004.

Venkita Subramonian and Christopher Gill, "A Generative Programming Framework for Adaptive Middleware", *37th Hawai'i International Conference on System Sciences* (HICSS), January 2004. Won the Best Paper Award in the Software Technology Track.

Michael Frisbie, Douglas Niehaus, Venkita Subramonian and Christopher Gill, "Group Scheduling in Systems Software", *Workshop on Parallel and Distributed Real-Time Systems* (WPDRTS), April 2004.

Christopher Gill, Venkita Subramonian, Jeff Parsons, Huang-Ming Huang, Stephen Torri, Douglas Niehaus, and Douglas Stuart, "ORB Middleware Evolution for Networked Embedded Systems", *Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems* (WORDS), January 2003.

Short Title: Principled Composition of Middleware Subramonian, D.Sc. 2006