Using Model Checking to Detect Deadlocks in Distributed Object Systems*

Nima Kaveh

Dept. of Computer Science, University College London
London WC1E 6BT, UK
n.kaveh@cs.ucl.ac.uk

Abstract. We demonstrate how the use of synchronization primitives and threading policies in distributed object middleware can lead to deadlocks. We identify that object middleware only has a few built-in synchronization and threading primitives. We suggest to express them as stereotypes in UML models to allow designers to model synchronization and threading of distributed object systems at appropriate levels of abstraction. We define the semantics of these stereotypes by a mapping to a process algebra. This allows us to use model checking techniques that are available for process algebras to detect the presence or absence of deadlocks. We also discuss how the results of these model checks can be related back to the UML diagrams.

Keywords: Software Architecture, Object Middleware, Model Checking

1 Introduction

An increasing number of applications now use a distributed system architecture. If designed properly, these architectures can be more fault-tolerant due to replicated components, can achieve better response times if user interface components are executed on powerful desktop machines or workstations, and they may achieve cost-effective scalability by using several relatively cheap hosts to execute replicated components rather than one central server or mainframe, which is usually more expensive. The construction of such distributed systems by directly using network operating system primitives, such as TCP or UDP sockets, is rather involved. To reduce this complexity, software engineers use *middleware* [5], which resolves the heterogeneity between distributed hosts, the possibly different programming languages that are being used in the architecture and provides higher level interaction primitives for the communication between distributed system components. There are many different forms of middleware, including transaction monitors, message brokers and distributed object middleware, which encompasses middleware specifications such as the Object Management Group's Common Object Request Broker Architecture (CORBA), Microsoft's Component Object Model (COM) or Java's Remote Method Invocation (RMI). We note that distributed object middleware offers the richest support to application designers and incorporates primitives for distributed transaction management and asynchronous message passing.

* This paper is an extended version of [6]

From the set of distributed object middleware approaches, we concentrate on CORBA [5] in this paper because it offers the richest set of synchronization and threading primitives.

An example scenario is used throughout this paper, which we will use to demonstrate our ideas and methods. This example involves the remote monitoring of patients which have been retired from the hospital to their homes. Sensor devices are attached to patients and information is communicated between the sensor devices and a central server in a health care centre. Additionally each patient is equipped with an alert device used in case of an emergency. This example is an inherently distributed system. The different approaches have in common that they enable distributed objects to request operation executions from each other across machine boundaries. We refer to this primitive as an *object request*. For this example, we will use object requests to pass diagnostic information about patients that are gathered by sensor devices to a centralized database where the diagnostic data are evaluated, and if necessary alarms are generated.

Distributed objects that reside on different hosts are executed in parallel with each other. In our example, this means that several different patient monitor hosts gather patient data at the same time. To handle the situation where several of them send data concurrently to a server, distributed object middleware supports different threading policies, which determine the way in which the middleware deals with concurrent object requests. A *single-threaded* policy will queue concurrent requests and execute them in a sequential manner, whereas a *multi-threaded* policy can deal with multiple requests concurrently. A common method of implementing multi-threaded policies is to define a thread pool, from which free threads are picked to process incoming requests and requests are queued if the pool is exhausted.

Object requests need to be synchronized, because client and server objects may execute in parallel. Object middleware support different synchronization primitives, which determine how client and server objects synchronize during requests. *Synchronous* requests block the client object until the server object processes the request and returns the results of the requested operation. This is the default synchronization primitive not only in CORBA, but also in RMI and COM. *Deferred synchronous* requests unblock the client as soon as it has made the request. The client achieves completion of the invocation as well as the collection of any return values by polling the server object. With a *oneway* request there is no value returned by the server object. The client regains control as soon as the middleware received the request and does not know whether the server executed the requested operation or not. *Asynchronous requests* return control to the client as soon an invocation is made. After the invocation the client object is free to do other tasks or request further operations. The result of the method invocation is returned in a call back from the server to the client. We note that CORBA supports all these primitives directly. In [5], its shown how the CORBA primitives can be implemented using multiple client threads in Java/RMI and Microsoft's COM.

The main contributions of this paper are firstly an identification of an important class of liveness problems in distributed object systems. We use the example scenario to demonstrate how particular combinations of synchronization primitives and threading policies in CORBA can lead to deadlocks. Secondly, we exploit the fact that object middleware only has a few built-in synchronization and threading primitives and express these as

stereotypes in dynamic UML models. Thirdly, we define the semantics of these stereotypes by mapping stereotyped UML models to a process algebra. Finally, we show how model checking techniques available for these process algebra notations are able to detect the possibility of deadlocks and how their results can be related back to the UML models.

Application developers need to verify their design specifications for absence of any deadlock situations. We aim to develop a CASE tool that takes in such design specifications, from which we generate a process algebra specification which is then analysed for deadlock. This approach has the advantage of detecting deadlocks in the design stage of development compared to the traditional way of attempting it during the later testing phase. Early indications of potential deadlock situations will make the process of design and implementation modifications more efficient.

In Section 2, we will define UML stereotypes to express both the threading policies and the synchronization primitives of distributed object middleware. In Section 3, we explain informally how a deadlock occurs in the running example. We then define the semantics of our threading and synchronization stereotypes using FSP, a process algebra representation [9] in Section 4. We explain in Section 5 how we use this semantics definition to generate an FSP process model from a UML Sequence Diagram. Section 6 discusses how compositional reachability analysis can be used to check for presence or absence of deadlocks in our Sequence diagrams. Section 7 concludes our paper by summarizing the main results and indicating future directions of this research.

2 Modelling Distributed Object Interactions

Rather than proposing to use new or complex notations and tools we have chosen the Unified Modelling Language to model object interactions and their class structures. UML is widely accepted and used in industry and allows us to enrich its notation to accomodate the extra semantics required for model checking. In this section we will look at modelling the described example, with an aim of using it for deadlock detection later in the paper.

Stereotypes provide designers with the means of augmenting basic UML models to include the semantic information required to model the synchronization behaviour in an application design. We have separated the stereotypes into two main groups. The first group deals with the synchronization primitives used by a client to request the services of a server object. The second group involves threading policies used on the server-side. These policies determine how server objects deal with multiple concurrent requests.

The ⟨⟨`Synchronous`⟩⟩ stereotype represents the synchronous request primitive request, whilst the ⟨⟨`DeferredSynchronous`⟩⟩ stereotype is used to indicate a deferred-synchronous request being made on a server object. ⟨⟨`Asynchronous`⟩⟩ is used to indicate an asynchronous client request and a ⟨⟨`OneWay`⟩⟩ stereotype represents a oneway request. Similarly on the server-side, we have defined the ⟨⟨`singleThreaded`⟩⟩ streotype to indicate that a particular server object uses a single threaded policy to deal with incoming service requests and the ⟨⟨`multiThreaded`⟩⟩ stereotype shows that the server object handles multiple service requests by using multi-threading techniques.

The class diagrams in Figures 1 and 2 show the main object types involved in gathering patient data and communicating it to a health care centre database. Note that the design diagrams mainly address distributed communication. Issues such as the user interface need to be looked at separately and are of no concern here.

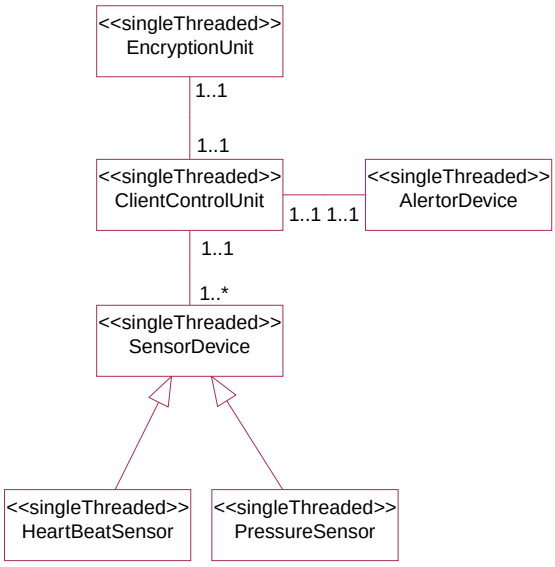

Fig. 1. UML class diagram of the Client

`SensorDevice` is an abstract type for all types of medical sensor devices attached to patients. `HeartBeatSensor` and `PressureSensor` are two concrete object types inheriting from the `SensorDevice` class. The `AlertorDevice` represents the device that a patient activates to get medical attention. The `ClientControlUnit` is used by sensor devices to send updates and alert messages to the health care centre server. `EncryptionUnit` ensures a secure communication channel as well as providing a non-repudiation service, which will be used when charging patients for services.

All patient data are stored in a central database at the health care centre. All incoming data from patients are logged and appropriate updates are made to the central database. `ServerControlUnit` is the main co-ordinator class on the server side. This type is responsible for communicating information between the patients and the health care centre, and its capable of servicing several patients simultaneously. In order to obtain this parallelism, it uses a multi-threading technique. The `DBThreadPool` represents a fixed-size collection of `DBDriverThread` objects, which it manages. Each time the `ServerControlUnit` receives a message from a patient it asks the `DBThreadPool` object to provide a free `DBDriverThread`, to which it delegates the task of processing the message. The delegated `DBDriverThread` object makes any required amendments to the central database and sends back any messages through the `ServerControlUnit`. Upon completion of

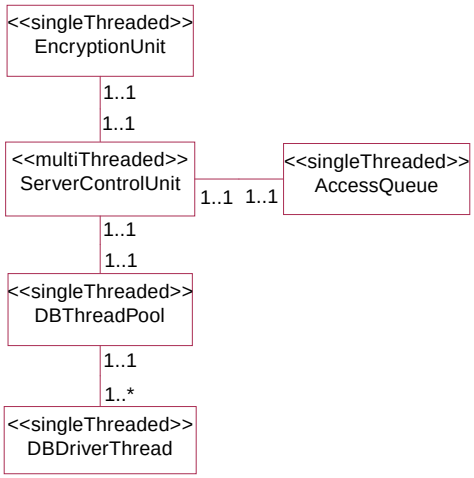

Fig. 2. UML class diagram of the Server

a task the `DBDriverThread` object reports back to the `DBThreadPool` objects and is deemed free once again. In the case of having no free `DBDriverThread` objects the `ServerControlUnit` will store all new incoming messages in the `AccessQueue` object. As the `DBDriverThread` objects start becoming free again, messages can be dequeued from the `AccessQueue` object and serviced, in a first-in-first-out order. In the event of all `DBDriverThread` object being occupied and the `AccessQueue` being at full capacity the `ServerControlUnit` will have to reject any messages which it receives.

Class diagrams are used to describe the structure and hierarchy in a design, thus containing static information. Whilst a sequence diagram represents a given scenario of how instances of classes interact with each other, thus containing dynamic information. The sequence diagram in Figure 3 describes the interactions of a `HeartBeatSensor` object to update information of a health care centre. Due to lack of space the `EncryptionUnit` has not been included in the sequence diagram, but its exemption does not alter the behaviour.

3 Deadlock in Distributed Object Systems

The source of a deadlock is often a cyclic wait-for relation between communicating components. The complex communication patterns between software components and a need to control the way their shared resources are accessed via methods such as mutual exclusion, gives way to deadlock vulnerabilities. This coupled with the inherent parallelism present in distributed systems, makes deadlock situations a likely and difficult problem to resolve. In fact, the default synchronization primitive and threading policy used in middleware systems, namely the synchronous request and the single threaded policy, are the most likely combination to bring about deadlock.

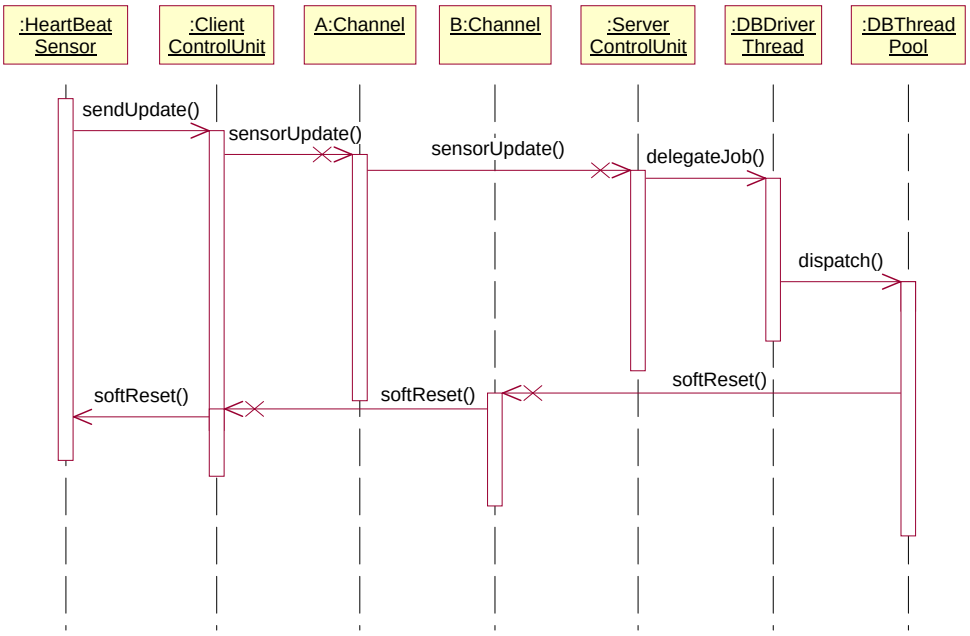

Fig. 3. Sequence diagram of a routine sensor update

The sequence diagram shown in Figure 3 actually results in a deadlock. First the `HeartBeatSensor` device sends an update through the `ClientControlUnit`, in the form of a synchronous request. Upon arrival on the server-side a `DBDriverThread` instance is assigned to deal with this message. Control is not returned back to the client until the `DBDriverThread` has finished processing the request. The `DBDriverThread` concludes that a soft reset of the sensor device is required and so it sends the reset command by invoking a synchronous request to the `HeartBeatSensor` object. Thus we have a case where the Client is blocked waiting for a response from the `DBDriverThread` and vice versa, thus causing a deadlock chain. This deadlock is not easily spotted, because as mentioned before the threading behaviour is determined at a type-level of abstraction and the synchronization behaviour is modelled at an instance level of abstraction. Only the combined knowledge of the two allows designers to consider the liveness issues. In order to demonstrate the idea we kept the interaction small. But the reader should note that such a detection would have been a lot more difficult in a real world industrial case, where the number of objects involved in an interaction may be considerably larger.

Deadlocks are inherently difficult to detect due to the large number of factors affecting the probability of their occurrence. Factors such as varying hardware resources and a wide range of possible user inputs creates a large number of scenarios to run an application. The conventional testing approach creates a test case for each of the likely scenarios in which the application is thought to be used under. The test cases are then executed and their results are compared with predefined expected results. Moreover, dis-

tributed applications make the task of testing more difficult by adding new dimensions of complexity. Prime examples of such complexities are hardware heterogeneity, a lack of global memory and physical clock and absence of bounds on transmission delays. We argue that the exponential growth in likely scenarios makes the conventional methods of testing much less effective and scalable. We argue that model checking provides a suitable method of overcoming this complexity dilemma as well tackling it with a rigour and thoroughness that cannot be expected from a human being.

4 Semantics of Stereotypes

A process algebra was chosen to define the semantics of the stereotypes ahead of alternatives such as denotational and axiomatic models, since it provides a more powerful mathematical model of concurrency. Process algebra operators offer direct support for modelling the inherent parallelism in distributed systems. The syntax allows for hierarchical description of processes, a valuable feature for compositional reasoning, verification and analysis.

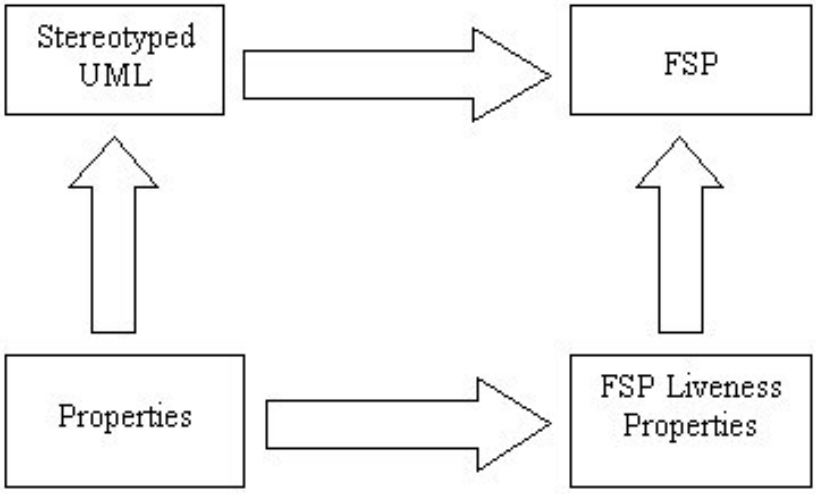

Fig. 4. Relation of FSP to design modelling

Figure 4 further demonstrates the reasoning behind choosing a process algebra for modelling the semantics of stereotypes. We are specifically referring to the FSP [9] process algebra. We would like to generate FSP specification from stereotyped UML models. There are liveness properties which the designer would like to have in these models such as deadlock safety, which are directly supported by the liveness properties experessed in FSP.

The CORBA Notification Service [13] uses an architectural element called an Event Channel which allows messages to be transferred between suppliers and consumers of

events. This service offers added capabilities such as being able to choose a level of Quality of Service and event filtering at the server-end. All client/server interactions in Figure 3 are taking place through such Channel objects. The generated FSP must exactly follow the semantic behaviour of the synchronization primitives and threading policies as outlined in Section 1.

4.1 Synchronization Primitives

The process algebra model in Figure 5 defines the ⟨⟨Synchronous⟩⟩ stereotype semantic of requests. The `Client` process engages in an action `SendRequest` and does not return until it receives a reply using the `ReceiveReply` action. By using relabelling we have synchronized the Client `SendRequest` with the Channel `ReceiveRequest` and the Client `ReceiveReply` with the Channel `SendReply`. So by making the Channel process engage in a SendReply action only after receiving a reply from the server, we define the ⟨⟨synchronous⟩⟩ stereotype.

```
Client=(SendRequest->ReceiveReply->Client).

Channel=(ReceiveRequest->RelayRequest->ReceiveReply->
SendReply->Channel).

||System=(c:Client || a:Channel)
/{c.SendRequest/a.ReceiveRequest,
  c.ReceiveReply/a.SendReply}.
```

Fig. 5. Process Algebra Definition of Synchronous Stereotype

The process algebra model in Figure 6 defines the ⟨⟨DeferredSynchronous⟩⟩ stereotype request semantic. The `Client` process invokes a request by engaging in action `push_sendRequest` which is synchronized with the `push_ReceiveReply`. The `WaitTime` constant defines the number of time units that the `Client` process continues executing before blocking to receive any results from the server. The `Client` is unblocked when the `Channel` process engages in action `push_sendReply`, which is called only when the server returns a result to the `Channel`.

4.2 Threading Policies

The FSP representation in Figure 7 defines the semantics of a server that uses a thread pool policy to handle multiple concurrent requests. The total number of slave threads and queue slots are specified as constants at the beginning. The server-side is composed of four processes, representing the slave thread, thread pool, queue and the server. The processes have synchronization points where they share the same action name. The `Server` process uses two variables to keep track of the current size of the queue and the number of threads currently in use. The server `ReceiveRequest` action indicates the arrival of a

```
const WaitTime=3
range T = 0..WaitTime

Client = (push_SendRequest->Client[0]),
Client[i:T]= if (i<WaitTime) then (execute->Client[i+1])
     else (push_ReceiveReply -> Client).

Channel=(push_ReceiveRequest->push_SendRequest->push_ReceiveReply->
push_SendReply->Channel).

||System=(c:Client || a:Channel)
/{c.push_SendRequest/a.push_ReceiveRequest,
  c.push_ReceiveReply/a.push_SendReply}.
```

Fig. 6. Process Algebra Definition of the Deferred Synchronous Stereotype

client request, if there are any available threads the synchronised action `getFreeThread` is taken which starts the `ThreadPool` process. This further causes the `Thread` process to be initiated using the shared `delegateTask` action. Once the request has been serviced the responsible `Thread` process engages in a `ReceiveReply` action which is shared with the `Channel` process, causing the results to be sent back to the client. If the number of used has reached the maximum the server attempts to add the message to the queue. This `addToQueue` succeeds if there are free queue slots left, otherwise the message is being rejected.

5 Generating FSP Models From UML Diagrams

We have identified a fixed number of synchronization primitives and threading policies used in mainstream object-oriented middleware systems. From these we obtain a fixed number of combinations in which they can be formed. We have defined the FSP specification for the semantics of each synchronization primitive and threading policy as demonstrated in section 4. The CASE tool will take as input, UML models enriched with stereotypes and translate them into a FSP specification. In order to achieve this we have to absorb information from two levels of abstraction, namely the type level and the instance level. The threading behaviour are specified in class diagrams with the aid of stereotypes whereas the synchronization behaviour is modelled in the interaction diagrams. The interaction between clients and server objects will involve a combination of synchronization primitives and threading policies. Thus the corresponding FSP specification will need to be formed from combining specification of a specific synchronization primitive with that of a threading policy. For example the FSP specification for the interaction between the `Channel` object `A` and the `ServerControlUnit` object in Figure 3 is formed by combining the specification in Figures 5 and 7. XMI will be used as the intermediate form, for the transition of input UML models into FSP specification. Research implementations for the UML to XMI transition [12] are well under progress and will benefit us.

```
const PoolSize=16
const QueueSize = 10
range T=0..PoolSize
range Q=0..QueueSize

Channel=(
  push_ReceiveRequest->push_SendRequest->push_ReceiveReply
  ->push_SendReply->Channel).

Thread=(delegateTask->taskExecuted->push_ReceiveReply->Thread).

ThreadPool = ThreadPool[0],
ThreadPool[i:T] = if (i<PoolSize) then
                     (getFreeThread->delegateTask->ThreadPool[i+1]
                    | taskExecuted -> ThreadPool[i-1])
  else (noFreeThreads -> ThreadPool[i]).

Queue = Queue[0],
Queue[j:Q] = if (j<QueueSize) then (
        inspectQueue-> if(j>0) then (dequeueMessage-> Queue[j-1]
                                   | addToQueue[j] -> Queue[j+1])
                       else (addToQueue[j] -> Queue[j+1]))
              else (rejectMessage -> Queue[j]).

Server = Server[0][0],
Server[i:T][j:Q]=(
  push_ReceiveRequest->
  if (i<PoolSize) then (
     getFreeThread-> Server[i+1][j])
  else
     (noFreeThreads->
     if (j<QueueSize) then (addToQueue[j]->Server[i][j+1])
     else (rejectMessage-> Server[i][j]))).

||System=(a:Channel||s:Server||s:ThreadPool||s:Thread||s:Queue)
        /{a.push_SendRequest/s.push_ReceiveRequest,
          a.push_ReceiveReply/s.push_SendReply}.
```

Fig. 7. Semantics Definition of ThreadPool Stereotype

6 Detecting Deadlocks By Model Checking

Once we have derived the FSP specification, we can use a model checker to do an exhaustive search for deadlocks. The Labelled Transition System Analysis tool that is available for FSP performs a compositional reachability analysis [2] in order to compute the complete state space of the model. This tool operates by mapping the specification into a Labelled Transition System [11]. A deadlock is detected by looking for states with ingoing but no outgoing transitions.

In the case of a deadlock detected, the LTSA will provide us with a trace of actions leading to the deadlock. From this trace we can single out the starting and ending link in the deadlock chain. In FSP terminology these links are processes and within each process we can find the actual action statement leading to deadlock. Figure 8 shows the output produced by the LTSA when processing the FSP specification of the example we have been discussing through out this paper. This FSP specification is formed by combining the specifications in Figures 5 and 7. As you can see the composition time is fairly quick, however the state space of the output is very large and its rate of growth is well above a linear relationship.

```
State Space:
 4 * 4 * 4 * 385 * 33 * 9 * 21 * 3 = 461039040
Composing
 potential DEADLOCK
States Composed: 10 Transitions: 9 in 10ms
```

Fig. 8. Output of the LTSA for the discussed example

7 Related Work

Process algebra representations, such as CSP [8], CCS [10], the π-calculus [11] or FSP [9] can be used to model the concurrent behaviour of a distributed system. Tools, such as the Concurrency workbench [3] or the Labelled Transition System Analyzer available for FSP can be used to check these models for violations of liveness or safety properties. The problem with both these formalisms and tools is, however, that they are difficult to use for the practitioner and that they are general purpose tools that do not provide built-in support for the synchronization and activation primitives that current object middleware supports.

Many architecture description languages support the explicit modelling of the synchronization behaviour of connectors by means of which components communicate [14]. Wright [1], for example uses CSP for this purpose. A main contribution of [4] is the observation that connectors are most often implemented using middleware primitives. In our work, we exploit the fact that every middleware only supports a very limited set of connectors, which can be provided to practitioners as stereotypes that are very easy to use.

In [7] CCS is used to define the semantics of CORBA's asynchronous messaging. The paper however, fails to realize that the synchronization behaviour alone is insufficient for model checking as deadlocks can be introduced and resolved by the different threading policies that the object adapters support.

8 Acknowledgements

I would like to thank Wolfgang Emmerich for his continual feedback and useful suggestions. I am also indebted to Jeff Magee for expressing his views on an earlier version of this paper.

References

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.
2. S.-C. Cheung and J. Kramer. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, 1999.
3. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.
4. E. di Nitto and D. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proc. of the* 21^{st} *Int. Conf. on Software Engineering, Los Angeles, California*, pages 13–22. ACM Press, 1999.
5. W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Apr. 2000.
6. W. Emmerich and N. Kaveh. Model Checking Distributed Objects. In B. Balzer and H. Obbink, editors, *Proc. of the* 4^{th} *International Software Architecture Workshop, Limerick, Ireland*, 2000. To appear.
7. M. Gaspari and G. Zavattaro. A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service. In *Proceedings of the* 13^{th} *European Conference on Object-Oriented Programming, ECOOP'99*, volume 1628 of *Lecture Notes in Computer Science*, pages 495–518. Springer, 1999.
8. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
9. J. Magee and J. Kramer. *Concurrency: Models and Programs – From Finite State Models to Java Programs*. John Wiley, 1999.
10. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1995.
11. R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
12. C. Nentwich, W. Emmerich, A. Finkelstein, and A. Zisman. Browsing Objects in XML. Research Note RN/99/41, University College London, Dept. of Computer Science, 1999.
13. Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.0*. 492 Old Connecticut Path, Framingham, MA 01701, USA, July 1995.
14. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.