Model Checking Distributed Objects

Wolfgang Emmerich and Nima Kaveh
Dept. of Computer Science, University College London
London WC1E 6BT, UK
{w.emmerich|n.kaveh}@cs.ucl.ac.uk

Abstract

We demonstrate how the use of synchronization primitives and threading policies in object middleware can lead to deadlocks. We identify that object middleware only has a few built-in synchronization and threading primitives and suggest to express them as stereotypes in UML models. We define the semantics of these stereotypes by a mapping to a process algebra. Finally, we apply model checkers to this process algebra notation and show that we are able to detect the possibility of deadlocks that can then be related back to the UML models.

1 Introduction

The increasing demand for distributed applications in a wide variety of fields has increased the commercial attention to middleware. Like so many other technologies, middleware has grown from being a research topic into a maturing commercial technology. A large number of distributed systems are built using *middleware*, which shields the use of networking protocols from the application programmer. There are different categories of middleware, which include transactional, message-oriented and object middleware. Of these categories we are particularly interested in object middleware, as it provides the highest level of abstraction by allowing a client object to request the execution of an operation from a server object. We refer to this primitive as an *object request*. Object requests are supported by different object middleware, which include OMG/CORBA [13], COM [7] and Java/RMI [9].

Objects that are distributed across different machines have the potential to be executed concurrently. A client object that resides on one machine executes concurrently with the server object that resides on a different machine. Moreover, there may be several different client objects that may request operation executions from a server concurrently. This concurrent execution poses interesting questions about how the synchronization between client and server objects is achieved. Distributed systems can exhibit true parallel behaviour, due to network nodes physically operating in parallel over a range of geographical locations. Concurrent and parallel execution, however, lead to a number of potential problems, such as deadlocks, livelocks, safety property violations.

The main contributions of this paper are firstly an identification of an important class of liveness problems in distributed object systems. We demonstrate how the use 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 apply model checkers to these process algebra notation and show that we are able to detect the possibility of deadlocks that can then be related back to the dynamic UML model.

In the next section, we discuss the synchronization and threading primitives that are supported by current object middleware and demonstrate how they can lead to deadlocks. In Section 3, we define UML stereotypes for all CORBA synchronization primitives and threading policies and define

the semantics by explaining how stereotyped UML models can be mapped to Finite State Processes (FSP) [10], the process algebra notation we use. In Section 4, we show how process algebras can be used to detect these liveness problems. We discuss related work in 5. The final section includes the summary and an outline of our future plans for extending this idea further.

2 Liveness Problems in Object Middleware

Object middleware support different synchronization primitives. They 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], we show how the CORBA primitives can be implemented using multiple client threads in Java/RMI and Microsoft's COM.

Threading policies determine the way in which an object adapter deals with 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.

Unlike in centralized systems, where recursion does not cause any liveness problems (provided that it terminates), particular combinations of synchronization primitives and threading policies in recursive object requests may lead to deadlocks. As an example, consider two distributed objects A and B. If both A and B use a single-threaded object adapter and A requests a synchronous operation execution from B during which B requests a synchronous operation from A then the two objects are in a deadlock. The deadlock, however, will not occur, if either of the two objects use a multi-threaded policy or if either of the object requests are asynchronous.

3 Modelling Distributed Object Communication

Deadlocks that involve more than a few objects are difficult to detect manually and therefore developers need automated tool support. We aim to provide this support at an early stage in the development process and also want to enrich notations and tools that designers use in industrial practice rather than propose the use of new or complicated notations and tools. We therefore discuss UML stereotypes that can be used to express synchronization primitives of object requests and the use of threading policies in object adapters.

We define two stereotypes ⟨⟨`multiThreaded`⟩⟩ and ⟨⟨`singleThreaded`⟩⟩. As the threading policy is common to all instances of a distributed object type, we use these stereotypes in conjunction with classes in class diagrams. Moreover, we define ⟨⟨`synchronous`⟩⟩, ⟨⟨`deferredSynchronous`⟩⟩, ⟨⟨`oneway`⟩⟩ and ⟨⟨`asynchronous`⟩⟩ as stereotypes for synchronization primitives that are used for an object request. As these are defined for individual object requests, we attach them to messages in sequence or collaboration diagrams. In accordance with established UML practice, we draw the ⟨⟨`synchronous`⟩⟩ stereotype as an arrow with a cross and the ⟨⟨`asynchronous`⟩⟩ stereotype with only a half arrow head.

An example of the use of these stereotypes is shown in Figure 1. The figure shows the communication of a client with a server object through two event channels. The first channel is used to push some data from the client to the server and the second channel is used to push data from the server to the client. The left-hand side of the figure shows a class diagram where stereotypes are used to define that distributed client, server and channel objects are to be executed in a single-threaded manner. The right-hand side shows messages with stereotype ⟨⟨`synchronus`⟩⟩ that determines that the client object that executes an object request is blocked while the server executes the requested operation.

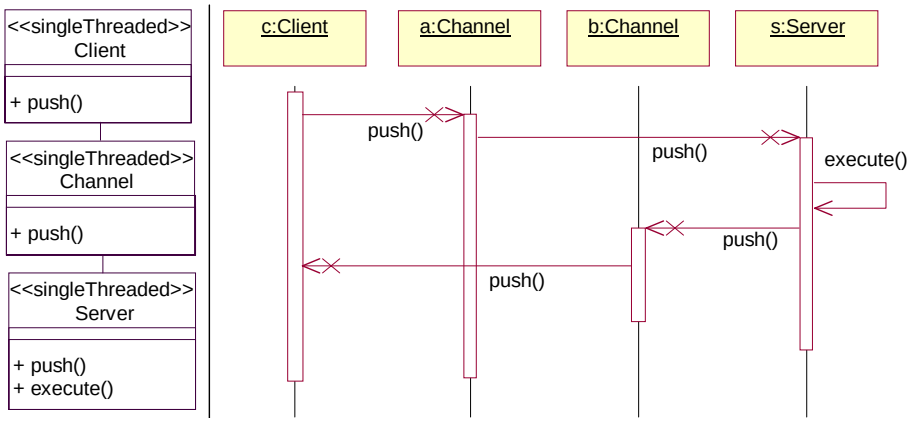

Figure 1: Use of Threading and Synchronization Stereotypes

We note that the above sequence diagram depicts an interaction that leads to deadlock. This deadlock is not easily spotted, because the distributed object threading behaviour is determined at a type level of abstraction, i.e. in a class diagram, and the synchronization behaviour is modelled at an instance-level of abstraction, that is in an interaction diagram. Only the combined knowledge of threading policies of classes and synchronization primitives that are used for object requests allows designers to consider the liveness. Also the interactions in industrial distributed object applications are much more complex than this simple example and thus it is usually impossible to determine deadlock freedom manually. We thus aim to support designers with automated model checks.

4 Model Checking

The stereotyped class and interaction diagrams provide sufficient information to automatically derive a formal specification of the synchronization and threading behaviour. We use the FSP process algebra notation [10] for this purpose, mainly because it is supported by an easily accessible model checker. In this position paper we only show the result of this generation for the above example.

Figure 2 shows the FSP definition for the synchronization behaviour of the distributed object interactions that we showed in Figure 1. The first three statements define the threading behaviour of `Client`, `Channel` and `Server` objects. In particular they determine that requests are processed in the order they arrive in and that no concurrent threads are spawned upon request arrivals. The synchronization behaviour is then determined for each request by binding the sending of the request to the receiving of the request and by binding the sending of the reply to the receiving of the reply.

Once we have derived this process algebra 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. In our example, the labelled transition system analysis reveals the following result:

```
Client=(push_SendRequest->push_ReceiveReply->Client
       |push_ReceiveRequest->compute->push_SendReply->Client).

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

Server=(push_ReceiveRequest->execute_call->push_SendRequest->
        push_ReceiveReply->execute_return->push_SendReply->Server).

||Test=(c:Client || a:Channel || b:Channel || s:Server)
        /{c.push_SendRequest/a.push_ReceiveRequest,
          c.push_ReceiveReply/a.push_SendReply,
          a.push_SendRequest/s.push_ReceiveRequest,
          a.push_ReceiveReply/s.push_SendReply,
          s.push_SendRequest/b.push_ReceiveRequest,
          s.push_ReceiveReply/b.push_SendReply,
          b.push_SendRequest/c.push_ReceiveRequest,
          b.push_ReceiveReply/c.push_SendReply}.
```

Figure 2: Process Algebra Defining Synchronization Behaviour

```
State Space:
 4 * 4 * 4 * 6 = 384
States Composed: 5 Transitions: 4 in 5ms
Trace to DEADLOCK:
c.push_SendRequest
a.push_SendRequest
s.execute_call
s.push_SendRequest
```

We have thus found a deadlock, which originates in the use of synchronous object requests and single threaded server objects throughout the example. The deadlock can be resolved, for example by executing the push operations in the client as a one-way operation, or by making the client multi-threaded.

5 Related Work

Process algebra representations, such as CSP [8], CCS [11], the π-calculus [12] or FSP [10] 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 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 [15]. 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 to practitioners as stereotypes that are very easy to use.

In [6] 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.

6 Summary and Further Work

We have demonstrated how the use of synchronization primitives and threading policies in object middleware can lead to deadlocks. We have identified that object middleware only has a few built-in synchronization and threading primitives and defined UML stereotypes for expressing them. We sketched how the semantics of these stereotypes can be defined by mapping them to a process algebra. Finally, we have shown how model checkers can be to detect deadlocks.

There are several directions for how we want to pursue these initial ideas. Firstly, we want to precisely define the mappings between the stereotypes and the process algebra in such a way that they can be automated by a tool. Secondly, we want to build such tool by building a compiler that translates UML models represented in XMI [14] into FSP. Finally, we want to find ways to express other safety and liveness properties in UML in order to then check them in their process algebra notation.

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] 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.

[7] R. Grimes. *DCOM Programming*. Wrox, 1997.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[9] JavaSoft. *Java Remote Method Invocation Specification*, revision 1.50, jdk 1.2 edition, Oct. 1998.

[10] J. Magee and J. Kramer. *Concurrency: Models and Programs – From Finite State Models to Java Programs*. John Wiley, 1999.

[11] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1995.

[12] R. Milner. *Communicating and Mobile Systems: the* π*-calculus*. Cambridge University Press, 1999.

[13] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.2*. 492 Old Connecticut Path, Framingham, MA 01701, USA, February 1998.

[14] Object Management Group. *XML Meta Data Interchange (XMI) – Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF)*. 492 Old Connecticut Path, Framingham, MA 01701, USA, Oct. 1998.

[15] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.