Deadlock Detection in Distributed Object Systems

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

ABSTRACT

The behaviour of a distributed system is largely determined by the use of synchronization primitives and threading policies of the underlying middleware. The inherent parallel nature of distributed systems may cause liveness problems, such as deadlocks and livelocks. An increasing number of distributed systems is built using object middleware. We exploit the fact that modern object middleware offers only a few built-in synchronization and threading primitives by suggesting UML stereotypes to represent each of these primitives in distributed object design. We define the semantics of the stereotypes using a process algebra. We use that semantics to translate UML diagrams into behaviourally equivalent process algebra representations and can then use model checking techniques to find potential deadlocks. The paper also shows how the model checking results can be related back to the original UML diagrams.

Keywords: Model Checking, Object Middleware, Process Algebra, UML

1. INTRODUCTION

The advances in networking technology are increasingly making the deployment of distributed system architectures a popular, sometimes even an essential option. The main advantages of distributed system architectures include increased overall system availability through better fault tolerance, parallel execution of an application on multiple hosts and a simplification of scalability.

The direct use of networking primitives or proprietary technologies for the development of distributed applications is no longer a viable option. Such approaches stifle application maintainability and ease of interoperability with other applications developed with other proprietary technologies. Instead, open object and component middleware technologies, such as CORBA [13] and Enterprise Java Beans [12], are rapidly becoming the preferred approach for the development of distributed systems.

These middleware approaches attempt to hide the complexity of distribution and aspire to provide software engineers with the ability to invoke operations on remote hosts in the same way as they would invoke local methods. While they succeed in many respects, there are some fundamental differences between local and remote method invocations [3]. One such difference is the inherent parallel execution of objects or components that reside on different machines. A local method call can recursively call itself, possibly indirectly via some other methods, and will not cause any problems as long as the recursion terminates at some stage. Recursion of distributed objects may however cause deadlocks.

In this paper, we investigate deadlock detection in-depth. We show that particular combinations of synchronization and threading policies may cause liveness problems in distributed systems that are built with object and component middleware. This is demonstrated through an example scenario. We aim to support the software engineer in detecting deadlocks in their distributed object designs. We exploit the fact that object and component middleware only offers a fixed number of client-side synchronization primitives and server-side threading policies. We suggest UML stereotypes to represent each of these primitives in distributed object designs. We define the semantics of the stereotypes using a process algebra. We use that semantics to translate UML diagrams into behaviourally equivalent process algebra representation and can then use model checking techniques to detect violations of liveness properties. We show how existing model checking tools can be applied to the generated process algebra for the detection of deadlocks. Finally, we demonstrate how model checking results can be related back to the original UML design model.

In Section 2, we present a scenario that exemplifies the problem. This scenario is used throughout the paper to demonstrate our approach. Section 3 gives details of how UML stereotypes are used to model the identified synchronisation characteristics of a given system and includes UML models of the example scenario. In Section 4, we define the semantics for the identified synchronization primitives and threading policies using a process algebra. Section 5 shows how the provided semantic definitions are used to generate a process algebra specification of UML statecharts and class diagrams. Section 6 discusses how we detect potential synchronization flaws in the given UML models using compositional reachability analysis [1]. An overview and comparison to related research is given is Section 7.

2. MOTIVATING SCENARIO

In order to aid convey and demonstrate the problem that we are addressing we present an example scenario for which we envisage the use of a distributed object or component middleware. This scenario is used throughout the paper to show how the system is modelled during the different stages of our approach and explain the results we obtain.

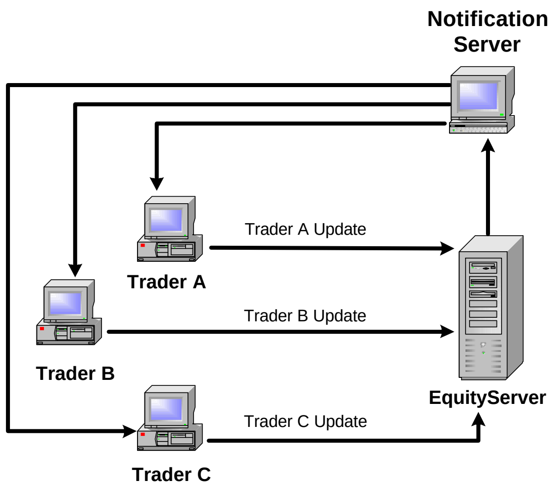

Figure 1: Market Trading Scenario

Figure 1 depicts a stock trading system, its main components and the communication channels between the components. To keep the scenario simple we concentrate on entities responsible for communication. We do not make any assumption about the underlying setup of the system. The three types of components are connected by a network and communicate via object-oriented middleware.

Market traders carry out transactions and monitor fluctuations in various stock prices. Triggered by changes in prices or external requests from customers to deal in particular stock, a `Trader` will enter a new transaction and send it to the `EquityServer`. Figure 1 shows three traders sending update¡s to the `EquityServer`. Note that the `Trader` entity could in reality consist of multiple components but for all intents and purposes of this scenario it is viewed as a simple entity that can send and receive information.

Upon receipt of trading information the `EquityServer` will carry out specific computations based on the received data and other sources, such as stock profiles stored in a database. At a certain point the `EquityServer` will complete a transaction and use this data to feed new price information to all traders. To do so, the `EquityServer` sends an updated price to the `NotificationServer`, which, in turn, publishes the price to registered traders. The delegation of the task to the notifications server simplifies the `EquityServer` and minimises coupling. We assume that all traders have registered with the `NotificationServer` during initialisation and that communication channels are already established.

Communication between all entities in the system follows the push model. In this model information flows in one direction and is initiated by the source. In our example the sink end always reacts by pushing information to the next entity. This creates recursion, whereby a `Trader` component calls an operation from the `EquityServer`, which calls an operation from the `NotificationServer` and this, in turn eventually calls back the `Trader` to notify a new price. If all these operations are called in a synchronous manner and servers are single threaded, we will reach a situation where all the components are blocked waiting the reception of information from one another, thus entering a deadlock. The occurence of this deadlock and systematic ways of detecting it serve as the motivation for our research.

3. MODELLING OBJECT INTERACTION

We use the Unified Modelling Language [15] for designing the static export interfaces of distributed object types and their dynamic object interactions. UML is widely accepted and deployed in industry and we hope to leverage its popularity to bring our research results into industrial practice.

Initially we chose UML class and interaction diagrams to model a given system [7]. This resulted in the system being represented at a type level of abstraction through class diagrams and an instance level of abstraction through interaction diagrams. The use of interaction diagrams limited us in obtaining only one specific interleaving of interactions between objects. This clearly did not take full advantage of the exhaustive search powers of model checking techniques. In this paper, we use UML statechart diagrams [5] rather than interaction diagrams to model the dynamic behaviour of distributed objects. Statecharts maintain the ability to model dynamic behaviour but because they model the behaviour at a type-level of abstraction they also hold all possible interleaving of object interactions in a given system.

The behaviour of distributed object interactions is governed by synchronization and threading policies. We note that current distributed object and component middleware systems support a fixed number of such synchronization and threading primitives. OMG's CORBA, Microsoft's Component Object Model (COM), Enterprise Java Beans (EJB) and Java Remote Method Invocation (RMI) all support synchronous invocations, which block the client until the server returns the result. CORBA also supports deferred synchronous, oneway and asynchronous invocations. Server objects, similarly, only support a small number of threading models. CORBA's Portable Object Adapter defines single-threaded behaviour, which would force a client to wait while a server object is busy processing another request and multi-threaded behaviour, which is often implemented by spawning new threads for requests or by selecting a thread from a thread pool. RMI supports directly only single threaded behaviour, but server programmers can use Java's threading primitives to spawn new threads when requests arrive. As the synchronization and threading behaviour is of great importance for the overall design of a distributed object system, we believe that they should be captured in static and dynamic design diagrams. CORBA provides a superset of the synchronization primitives and threading policies of COM, EJB and RMI. We subsequently define stereotypes for all the primitives that CORBA provides. These primitives can then be used during the design of non-CORBA objects, too. Our approach therefore caters for design and deadlock detection of all applications based on mainstream object middleware.

The ⟨⟨Synchronous⟩⟩ stereotype represents a synchronous request primitive, while the ⟨⟨DeferredSynchronous⟩⟩ stereotype is used to indicate a deferred-synchronous request being made on a server object. The ⟨⟨Asynchronous⟩⟩ stereotype is used to indicate an asynchronous client request, and a ⟨⟨OneWay⟩⟩ stereotype represents a oneway request. Similarly on the server-side, we defined the ⟨⟨singleThreaded⟩⟩ stereotype 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 concurrent service requests by using multiple threads. These primitives and policies will be detailed in Section 4.

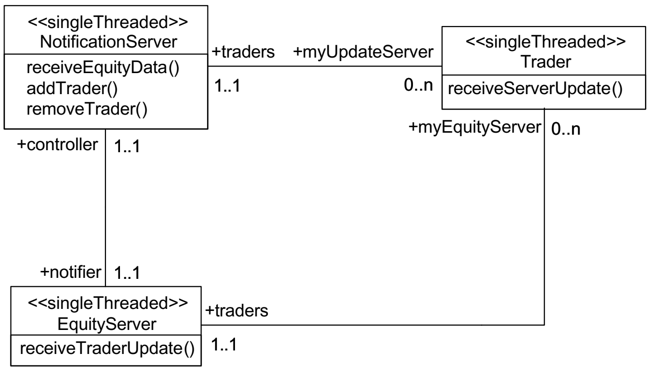

Figure 2: Class diagram of Market Trading Scenario

Server-side threading policies are defined statically for object types. We therefore model those in the class diagrams that capture the export interfaces of object types. As an example, Figure 2 shows a class diagram of the equity trading system. Each of the classes correspond to one of the three entities in the example scenario of Section 2. Each class is annotated with the ⟨⟨singleThreaded⟩⟩ stereotype, indicating that they handle one incoming request at a time. As discussed above, this is the default threading policy in mainstream middleware. Each class has a method responsible for receiving stock related information. This method is remotely invoked by an object of another class in order to push information to the recipient. Method `receiveTraderUpdate()` in the `EquityServer` class, for instance, is invoked remotely by an instance of the `Trader` class in order to pass any trading activity reports. Likewise, method `receiveServerUpdate()` of `Trader` is invoked by an object of type `NotificationServer` to pass the `EquityServer` updates.

Synchronization of remote operation invocations are a dynamic aspect and as such we define them in state diagrams. We use the above synchronization stereotypes in those transitions of statecharts whose actions correspond to remote operation invocations. The statechart of the `EquityServer` in Figure 3 initially starts in the `idle` state. When its operation is `receiveTraderUpdate` requested, it moves to state `update`. The action `notifier.receiveEquityData` that takes place whilst moving from `update` to `updates completed` is marked with a `<<synchronous>>` stereotype. This corresponds to the `receiveEquityData` method of the `NotificationServer` class in Figure 2. Notice that the action name contains the name of the association used in the class diagram. From this information we can deduct that an `EquityServer` object will request a remote synchronous operation from a `NotificationServer` object. Finally, the `EquityServer` goes back to the `idle` state causing a reply to be sent back to the `Trader` who sent the updates.

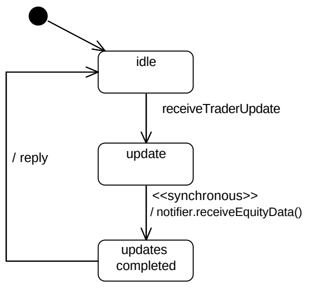

Figure 3: EquityServer Statechart

Figure 4 shows how the `NotificationServer` can register and unregister traders whilst in the `idle` state. Upon reception of update instructions from the `EquityServer` it moves into the `sending` state. It then continually sends updates via the `traders.receiveServerUpdates` action, until all traders have been notified. This action is marked with the `<<synchronous>>` stereotype. Similarly to the `EquityServer` case, we can deduce that instances of the `NotificationServer` class invoke the remote synchronous method `receiveServerUpdates` on `Trader` objects. The object re-enters the `idle` state upon updating all traders.

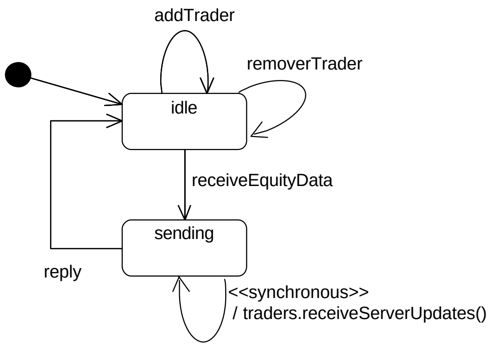

Figure 4: NotificationServer Statechart

Figure 5 shows the statechart for the `Trader` class. A trader processes a new transactions whilst in the `trading` state. It then sends the results of the trade to the `EquityServer` using the `myEquityServer.receiveTraderUpdate` action. This action is marked with a `<<synchronous>>` stereotype, indicating that the invocations made to instances of type `EquityServer` are synchronous. After replying to the `receiveTraderUpdate` event the object returns to state `idle`.

4. SEMANTICS OF STEREOTYPES

Process algebras represent mathematically rigorous frameworks for modelling concurrent systems of interacting processes. We have chosen process algebras for defining a formal semantics of our stereotypes over alternatives such as denotational and axiomatic models due to their more powerful model of concurrency. Process algebras allow for hierarchical description of processes, a valuable feature for compositional reasoning, verification and analysis. The particular

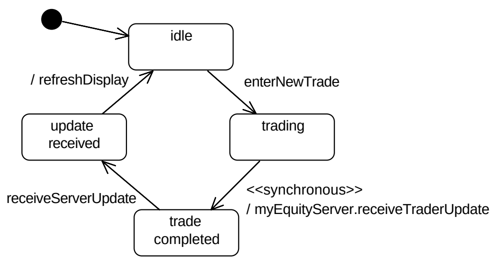

Figure 5: Trader Statechart

algebra that we have chosen for defining the semantics of the stereotypes are Finite State Processes [9] (FSP). We chose FSP because it is well-supported with a model checking tool.

There are two key concepts commonly used in our FSP specification of a system's synchronous primitive, namely synchronised actions and parallel composition. Each FSP process is composed of a set of actions that occur in a specified and fixed order. Parallel composition is used to describe a system with multiple concurrent processes, whereby the actions of the processes are interleaved. Therefore, whilst the actions of individual processes still occur in a fixed order, we obtain many different execution traces of the composite process. Note, that this directly reflects the concept of concurrent states in UML statecharts. Processes can be forced to perform actions simultaneously in a lock-step fashion via synchronized/shared actions. Actions with the same name are executed at the same time; this achieves synchronisation between concurrent processes. Actions with different names can be synchronized using the FSP relabelling mechanism. These concepts are demonstrated in the following subsections.

4.1 Synchronization Primitives

A *synchronous* request blocks 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. Figure 6 shows the FSP specification for a *synchronous* call. The Object Adapter (OA) intermediates between the client and the server process. The OA process receives requests sent by the Client process and relays them onto the Server process. `SynchInvocation` is a composite process made up of the parallel composition of the `Client`, `Server` and `OA` processes. It uses relabelling to synchronize the four actions of the `OA` process with the relevant actions in the `Client` and `Server` process. For example the `sendRequest` action of the `Client` process is synchronised with `OA`'s `receiveRequest` action, similarly the `OA`'s `relayReply` is synchronised with the `Server`'s `sendReply` action. This simply indicates that a client must have sent a request before the server sends back a reply. The overall execution of the composite process follows the order set in the `OA` process, therefore implementing a *synchronous* call.

With an *asynchronous* request control is returned to the client as soon as the invocation has been sent. Results of the invocation are returned to the client by a call-back mechanism invoked by the server. This means that the onus of directing the results to the client is now on the server. Figure 7 shows the FSP specification for the *asynchronous* invocation method. Similarly to the previous case the `OA` process mediates the synchronization of the `Client` and `Server` actions. However in this case, the client can engage in action `doOtherWork` infinitely often before it receives a call-back invocation from the server, via the OA. This is made possible by the FSP choice operator "|", which introduces a non-deterministic method of executing alternate actions.

```
Client=(sendRequest-> receiveReply-> Client).

OA=(receiveRequest->relayRequest->
    receiveReply->relayReply->OA).

Server=(receiveRequest->processRequest->
        sendReply->Server).

||SynchInvocation=(client:Client || serverOA:OA
                   ||server:Server)
/{client.sendRequest/serverOA.receiveRequest,
  client.receiveReply/serverOA.relayReply,
  server.receiveRequest/serverOA.relayRequest,
  server.sendReply/serverOA.receiveReply}.
```

Figure 6: Synchronous Stereotype Semantics

```
Client=(sendRequest->OtherExecutions),
OtherExecutions=(doOtherWork->OtherExecutions |
                 callBack->receiveReply->Client).

OA=(receiveRequest->relayRequest->
    receiveReply->relayReply->OA).

Server=(receiveRequest->processRequest->
        sendReply->Server).

||ASyncInvocation=(client:Client || serverOA:OA
                   ||server:Server)
/{client.sendRequest/serverOA.receiveRequest,
  client.callBack/serverOA.relayReply,
  server.receiveRequest/serverOA.relayRequest,
  server.sendReply/serverOA.receiveReply}.
```

Figure 7: Asynchronous Stereotype Semantics

Deferred synchronous invocations do not block the client immediately. The client object can engage in other actions for a duration of time. After the duration the client polls the server object to check for the availability of results. The polling is a blocking method causing the client to hold until a reply is sent from the server. Figure 8 shows the FSP specification for the *deferred synchronous* method invocation. The pre-polling duration has been set to 5 time steps. The `Client` process increments an internal timer and checks to see if it is at the polling threshold each time. After this point the specification behaves like a *synchronous* invocation and blocks until it receives a reply from the server.

A *oneway* method invocation does not block because there is no reply by the server. This offers an inexpensive way of invoking methods but offers no guarantees or indications as to whether the request has been received or processed by the server.

```
const WaitTime=5
range T = 0..WaitTime

Client = (sendRequest->Client[0]),
Client[i:T]= if (i<WaitTime)
             then (timeStep[i+1]->Client[i+1])
     else (receiveReply -> Client).

OA=(receiveRequest->relayRequest->
    receiveReply->sendReply->OA).

Server=(receiveRequest->processRequest->
        sendReply->Server).

||DeferredSynchInvocation=(client:Client||
                serverOA:OA|| server:Server)
/{client.sendRequest/serverOA.receiveRequest,
  client.receiveReply/serverOA.sendReply,
  server.receiveRequest/serverOA.relayRequest,
  server.sendReply/serverOA.receiveReply
}.
```

Figure 8: Deferred Synchronous Stereotype Semantics

4.2 Threading Policies

The primitives described in Section 4.1 were demonstrated in combination with a single threaded policy. The multi-threaded policy, expressed using the `<<multiThreaded>>` stereotype, allows for handling multiple requests simultaneously. There are several different methods of implementing this policy but all use the common principle of delegating the request handling to threads. Threadpools are a common implementation method, whereby new requests are delegated to threads drawn from a threadpool. Once the request has been processed the thread is returned to the threadpool and is declared available again. If all threads are busy at the time of a request arrival the request is put into a queue. In the situation where the queue is also full the request is discarded. Figure 9 defines the semantics of a server that uses a thread pool policy. 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 thread, threadpool, queue and the server. All server-side processes are composed with the same label so as to synchronise their action. 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 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`. If the number of used threads has not 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. GENERATION OF FSP

We observe that different combination of these primitives determine the synchronisation behaviour of their respective object implementations and thus the overall system. UML design models annotated with stereotypes defined in Section 4 can be broken down into a set of synchronisation

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

OA=(receiveRequest->relayRequest->
    receiveReply->sendReply->OA).

Thread=(delegateTask->taskExecuted->sendBackReply
        ->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]=(receiveNewRequest->
 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]))).

||MTSystem=(oa:OA||server:Server||
            server:ThreadPool||server:Thread||
            server:Queue)
    /{server.receiveNewRequest/oa.relayRequest,
      server.sendBackReply/oa.receiveReply}.
```

Figure 9: Semantics of Multi-Threaded Stereotype

primitive and threading policy combinations. As demonstrated in Section 4 we have developed a formal mapping for the semantics of each combination in the form of a process algebra. Therefore, by composing the derived FSP building blocks, we can generate a formal specification of the entire UML model.

We are currently developing a CASE tool to automate the generation of FSP from input UML models. The XML Metadata Interchange (XMI) [14] was chosen as the input format to the tool. XMI enables the encoding of MOF compliant meta models in XML and includes a type definition for representing UML models. All of the main modelling tools available have an XMI export facility. Moreover, the hierarchical Document Object Model representation of XMI documents provides an intuitive method of information extraction. Using the XMI document the CASE tool creates an object hierarchy of the UML elements present in the model. This hierarchy is formed in accordance with the UML meta-model specification. By traversing this object hierarchy we obtain the synchronization primitives and threading policies being used as well as the combinations in which they are used.

```
TRADER = (enterNewTrade->Trading),
Trading = (myEquityServer.receiveTraderUpdate -> receiveInvocationReply -> TradeCompleted),
TradeCompleted = (receiveServerUpdate-> UpdateReceived),
UpdateReceived = (refreshDisplay->TRADER).

OA=(receiveRequest->relayRequest->receiveReply->relayReply->OA).

EQUITYSERVER = (receiveTraderUpdate -> Update),
Update = (notifier.receiveEquityData-> receiveInvocationReply ->UpdatesCompleted),
UpdatesCompleted = (reply -> EQUITYSERVER).

NOTIFICATIONSERVER = (addTrader->NOTIFICATIONSERVER | removeTrader->NOTIFICATIONSERVER
                    | receiveEquityData->Sending),
Sending = (traders.receiveServerUpdates->receiveInvocationReply->Sending
         | finishedNotificationCycle->NOTIFICATIONSERVER).

||TradingSystem=(traderA:TRADER || traderB:TRADER || serverOA:OA || traderOA:OA || notifierOA:OA
                                ||server:EQUITYSERVER || notifier:NOTIFICATIONSERVER)
/{
  traderA.myEquityServer.receiveTraderUpdate/serverOA.receiveRequest,
  traderA.receiveInvocationReply/serverOA.relayReply,
  traderB.myEquityServer.receiveTraderUpdate/serverOA.receiveRequest,
  traderB.receiveInvocationReply/serverOA.relayReply,
  server.receiveTraderUpdate/serverOA.relayRequest,
  server.reply/serverOA.receiveReply,

  server.notifier.receiveEquityData/notifierOA.receiveRequest,
  server.receiveInvocationReply/notifierOA.relayReply,
  notifier.receiveEquityData/notifierOA.relayRequest,
  notifier.finishedNotificationCycle/notifierOA.receiveReply,

  notifier.traders.receiveServerUpdates/traderOA.receiveRequest,
  notifier.receiveInvocationReply/traderOA.relayReply,
  traderA.receiveServerUpdate/traderOA.relayRequest,
  traderA.refreshDisplay/traderOA.receiveReply,
  traderB.receiveServerUpdate/traderOA.relayRequest,
  traderB.refreshDisplay/traderOA.receiveReply}.
```

Figure 10: FSP specification of the motivating example

Figure 10 shows the generated FSP specification for the overall trading system. Each statechart described in Section 3 is mapped onto an FSP process. Furthermore, each of the states of the statecharts is modelled as a local process within its main process. Actions resulting from a transition from a state are modelled as FSP actions within each local process. The `OA` (Object Adapater) process operates in the same fashion as described in Section 4, synchronizing the actions for relaying requests and relaying back results of the invocation. Three instances of the `OA` process, `serverOA`, `notifierOA` and `traderOA`, are composed into the system since each of the three entities receive method invocations. This is made possible by using the FSP process labelling mechanism, which distinguishes the actions of instances of the same process from one another. If an action relates to a remote method invocation it is followed by a `receiveInvocationReply` action. For example, the local process `Update` of the `EQUITYSERVER` process in Figure 10 contains an action that corresponds to a remote method invocation that causes updates to be sent to a trader. The following `receiveInvocationReply` action is synchronized with the `notifierOA`'s `relayReply`. This forces the `TRADER` process to block until the `notifierOA` process has received a reply from the `NOTIFICATIONSERVER` process via the action `finishedNotificationCycle`. In this specification only two traders are modelled due to space restrictions for this paper.

6. MODEL CHECKING RESULTS

At this stage we have obtained a FSP specification of the entire UML model that we are analysing. The model checker included in the labelled Transition System Analyser (LTSA), which supports FSP [9] computes a labelled transition system of that specification, taking into considerations all possible interleaving of actions between instances, in our example, of the `EquityServer`, the `NotificationServer` and the `Trader` class. The LTSA describes components of a specification as Labelled Transition Systems [11] . The LTSA tool performs a compositional reachability analysis [1] to carry out an exhaustive search for violations of desired safety and liveness properties.

A deadlock is detected by searching for reachable states with no outgoing transitions, that would make the program stop once it entered one of these states. If a deadlock is detected, the LTSA produces a trace of actions that lead to the deadlock. Figure 11 shows the LTSA safety check output for the FSP code of the example shown in Figure 10. In this trace the trader process completes a new transaction and pushes the results to the `EquityServer` via a remote method invocation, blocking until it receives back a reply. Before processing the `Trader` request the `EquityServer` process pushes existing update information to the `NotificationServer` and blocks until it receives a reply. Upon receipt of the update information the `NotificationServer` makes blocking remote invocations of its own to the Trader process, before replying to the EquityServer. This last invocation completes a deadlock ring as all three processes are blocked waiting for one another. There are 19200 states in the state space. This grew to 96000 states when modelling three concurrent traders and 480000 states when considering 4 traders.

```
Compiled: TRADER
Compiled: OA
Compiled: EQUITYSERVER
Compiled: NOTIFICATIONSERVER
Composition:
TradingSystem=traderA:TRADER||traderB:TRADER||
     serverOA:OA||traderOA:OA||notifierOA:OA||
     server:EQUITYSERVER||
     notifier:NOTIFICATIONSERVER
State Space:
 5 * 5 * 4 * 4 * 4 * 4 * 3 = 19200
Analysing...
Depth 8 -- States: 25
           Transitions: 78
           Memory used: 3287K
Trace to DEADLOCK:
   traderA.enterNewTrade
   traderA.myEquityServer.receiveTraderUpdate
   traderB.enterNewTrade
   server.receiveTraderUpdate
   server.notifier.receiveEquityData
   notifier.receiveEquityData
   notifier.traders.receiveServerUpdates
Analysed in: 210ms
```

Figure 11: LTSA safety check output

We generate UML sequence diagrams from the property violation trace from the LTSA tool, to give feedback to the developer about potential problems in their design. Sequence diagrams allow us to show a specific set of interactions that can lead to a property violation in a presentable manner. Figure 12 is a sequence diagram derived from the deadlock trace detected by the LTSA. Local method calls are not shown as they have no effect on the inter-object interactions. The shape of the arrow heads are a visual sign of the <<synchronous>> stereotype. Note, how the name of the remote actions in Figure 11 contains the name of the associations defined in the class diagram of Figure 2.

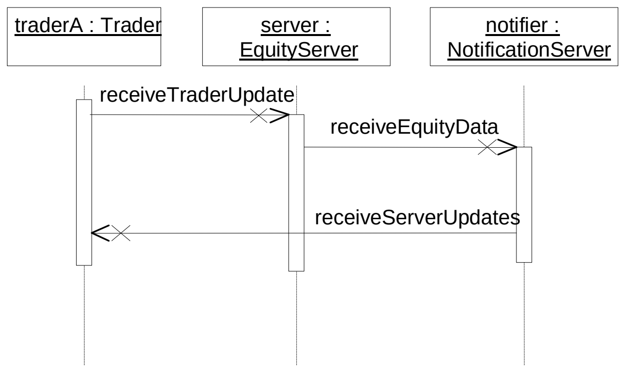

Figure 12: Sequence diagram showing deadlock

To conclude the motivating scenario we have been analysing, Figure 13 shows a new statechart for the Trader class. This is a composite state made up of two concurrent submachines, Trading and Updating. The Updating submachine deals with receiving updates from the NotificationServer, whilst the Trading submachine engages in trading transactions and pushes the results to the EquityServer. This new design allows a trader to wait for the EquityServer reply whilst receiving updates from the NotificationServer, thus freeing the system from deadlock.

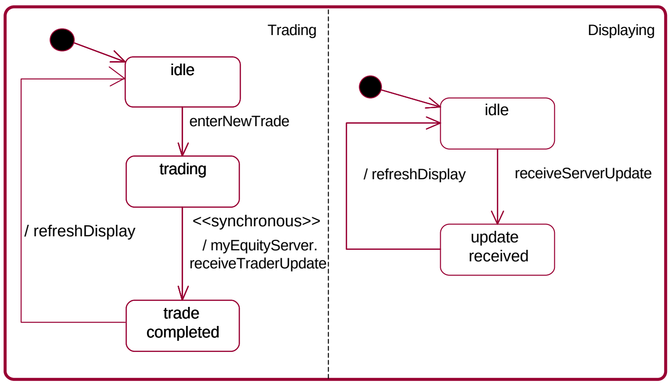

Figure 13: Concurrent Trader Statechart

7. RELATED WORK

Process algebra representations, such as CSP [6], CCS [10], the π-calculus [11] or FSP [9] can be used to model the concurrent behaviour of a distributed system. Tools like the Concurrency workbench [2] 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.

In [4], CCS is used to define the semantics of CORBA's asynchronous messaging. The paper, however, does not take into account that 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.

The work done in [8] is similar to our approach in that a formal specification is generated from UML design models. One of the assumptions made, however, is that each instance of the modelled class runs in a seperate process. This is not the case for object middleware as many server objects can run in the same process. Our stereotypes for single and multi-threaded server-side execution are closer to industrial reality.

In [16], FSP specifications are generated from an extended version of Message Sequence Charts (MSC) for the synthesis of system behaviour models. Whilst scenario-based specification is a suitable method for checking and communicating the key scenarios of a system it cannot be applied to detailed design models for the purposes of thorough validation and verification. The large number of key scenarios in a typical industrial case are too large to make this a scalable solution for design verification.

8. SUMMARY & FUTURE WORK

An increasing number of distributed systems is using distributed object and component middleware. This increase calls for support of developers by systematic software engineering methods and tools. Our research concentrates on synchronization behaviour in object-oriented middleware systems. The complexity increase introduced by the implicit and non-deterministic parallel behaviour of distributed systems creates a common source of error in middleware-based applications.

Our aim is to provide developers with a verification tool that can be used to check the integrity of their design with respect to specified safety and liveness properties. UML design models are annotated by developers using stereotypes representing the different synchronization behaviours in the model. We have developed formal mappings of these stereotypes enabling us to carry out verification on the generated formal specification. We therefore benefit from the rigorous model checking approach of formal methods whilst not burdening the developer with any new notation or knowledge.

We have identified a finite number of synchronization primitives used in object-oriented middleware systems. These consist of client-side synchronization primitives, which govern how clients make remote invocation calls and threading policies, which determine how servers handle method requests.

We plan to evaluate our approach with industrial models, which will be provided by our industrial collaborators. We also plan to extend this research to address different liveness and safety properties.

Acknowledgements

We are indebted to Cecilia Mascolo and Christian Nentwich for the valuable help they provided in improving the presentation of this paper.

9. REFERENCES

[1] S.-C. Cheung and J. Kramer. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–7, 1999.

[2] 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.

[3] W. Emmerich. *Engineering Distributed Objects.* John Wiley & Sons, Apr. 2000.

[4] M. Gaspari and G. Zavattaro. A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service. In *Proceedings of the* 13th *European Conference on Object-Oriented Programming, ECOOP'99*, volume 1628 of *Lecture Notes in Computer Science*, pages 495–518. Springer, 1999.

[5] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

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

[7] N. Kaveh. Model Checking Distributed Objects. In W. Emmerich and S. Tai, editors, *Proc. of the* 2nd *Int. Workshop on Distributed Objects, Davis, Cal, Nov. 2000*, volume 1999 of *Lecture Notes in Computer Science*, pages 116–128. Springer, 2001.

[8] J. Lilius and I. Paltor. A tool for verifying UML models. In *IEEE International Conference on Automated Software Engineering*, volume 14, 1999.

[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] R. Monson-Haefel. *Enterprise Javabeans.* O'Reilly UK, 1999.

[13] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.3.* 492 Old Connecticut Path, Framingham, MA 01701, USA, December 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] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley Longman, Reading, MA, USA, 1999.

[16] S. Uchitel and J. Kramer. A Workbench for Synthesising Behaviour Models from Scenarios. In *Proc. of the* 23rd *Int. Conf. on Software Engineering, Toronto, Canada.* ACM Press, 2001. To appear.