Validating Distributed Object and Component Designs*

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. Distributed systems are increasingly built using distributed object or component middleware. The dynamic behaviour of those distributed systems is influenced by the particular combination of middleware synchronisation and threading primitives used for communication amongst distributed objects. A designer may accidentally choose combinations that cause a distributed application to enter undesirable states or violate liveness properties. We exploit the fact that modern object and component middleware offer only a small number of underlying synchronisation primitives and threading policies. For each of these we define a UML stereotype and a formal process algebra specification of the stereotype semantics. We devise a means to specify safety and liveness properties in UML and again map those to process algebra safety and liveness properties. We can thus apply model checking techniques to verify that a given design does indeed meet the desired properties. We propose how to reduce the state space that needs to be model checked by exploiting middleware characteristics. We finally show how model checking results can be related back to the input UML models. In this way we can hide the formalism and the model checking process entirely from UML designers, which we regard as critical for the industrial exploitation of this research.

1 Introduction

Distributed software architectures prescribe the composition of software components intended to be deployed on a distributed system. There is an increasing trend of developing software applications based on distributed architectures. Increased overall system availability through better fault tolerance, parallel execution of an application and a simplification of scalability are some of the key motivators behind the popularisation of distributed architectures.

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 proprietary technologies. Instead, open object and

* This work is partially funded through EU project TAPAS (IST-2001-34069).

component middleware technologies, such as CORBA [26] and Enterprise Java Beans [22], 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 developers 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 [2]. 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. Due to the non-determinism introduced by components that execute in parallel, it is considerably more complicated to develop safe distributed applications than centralised applications.

Software engineers can now use these powerful middleware technologies for the implementation of distributed systems. The implementation support, however, needs to be complemented with appropriate architecture and design methods that address the new challenges that are introduced by the use of distributed object and component middleware. In particular, software engineers need support for reasoning about the correctness of a distributed object design that goes beyond the diagram drawing capabilities offered by current CASE tools.

In this paper we show that the use of particular combinations of client-side synchronisation primitives and server-side threading policies provided by most distributed object middleware may cause deadlocks as well as safety and liveness problems. We discuss a method to support the software engineer in detecting violations of desired system properties in their distributed object designs. We exploit the fact that object and component middleware standards and implementations only offer a fixed number of client-side synchronisation primitives and server-side threading policies. We suggest the use of 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 models and properties into behaviourally equivalent process algebra representations and can then use model checking techniques to detect any violations of the properties. Finally, we demonstrate how model checking results can be related back to the original UML design model. We present the tools that we have built in support of this method and evaluate the scalability of our validation technique.

In the next section, we discuss a scenario that we use throughout this paper to exemplify the problems that we address, as well as our solutions. 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. Section 4 shows how designers can express desired safety and liveness properties in UML for the design models to adhere to. Sections 5 and 6 use a process algebra to define the semantics for the identified synchronisation primitives and threading policies as well as user-defined safety and progress properties. In Section 7, we demonstrate the importance of tackling the state explosion problem and outline our efforts in that area. In Section 8, we show how deadlocks and safety property violations can be detected using reachability analysis as well as the use of efficient graph algorithms on the underlying state space for the detection of a restricted form of liveness properties. Section 9 shows the mechanism by which designers receive feedback from the verification process. We discuss the scalability of our approach in Section 10. Section 11 introduces the tool that we have built to support our approach, with focus on its design and architecture. Section 12 puts our work in context with related research in the field. Finally, we conclude in Section 13 and present future goals for this research.

2 Motivating Scenario

To aid the demonstration of this work, we discuss an example of a distributed software architecture, which we assume is implemented using object middleware technology. We refer to this scenario throughout the paper to demonstrate the key steps of our approach.

The example that we use is a stock trading system, which in practice is often distributed as different market participants interact from different locations with servers that are hosted by a stock exchange. In particular, traders need to interact with a component that executes orders when a transaction is completed. Every completed transaction at the same time determines a new price for a stock that needs to be communicated to all interested market participants.

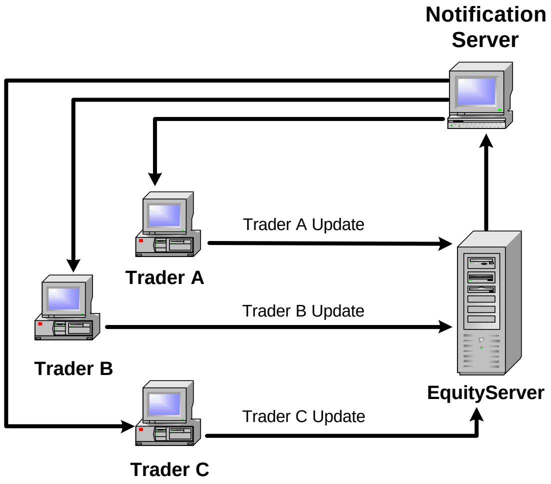

Fig. 1. Market Trading Scenario

Fig. 1 depicts the main components of the stock trading system and the communication channels between them. To keep the scenario simple, we only concentrate on the three types of entities responsible for communicating trade information, since it is these entities that determine the dynamic synchronisation behaviour of the application. We do not make any assumption about the infrastructure of the system, except that their hosts are connected by a network and that they 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 its results to the `EquityServer`. Fig. 1 shows three traders sending updates 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 processing the transaction results 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 notification 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 forwarding 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 it of 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.

Additionally, there are several domain specific properties that the designer may want the trading system to adhere to for the successful execution of the application. If we consider a closed market, prices are not changed in any other way than traders completing a transaction. This means that prior to any new prices being sent by the `NotificationServer`, traders need to send trade results to the `EquityServer` entity. Another desirable property is the guarantee that traders will be able to deal in stocks, no matter what the state of other components. Devising a means of representing these properties in a suitable notation and being able to verify a design model for such properties is the main theme of this paper.

3 Distributed Object Design

We use the Unified Modelling Language [29] 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. UML is a self-descriptive notation, in that its entities are defined via meta-model expressed in UML. The consequence of this approach is a lack of formal semantics for the notation, which is needed for rigorous verification of a design model. The UML standard also provides extension-mechanisms by which new semantics can be introduced into a model, whilst still remaining within the UML framework. This section describes how our approach uses the stereotype extension mechanism for embodying middleware specific information into UML design models.

Initially we chose UML class and interaction diagrams to model a given system [16]. 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 state diagrams [10] 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 synchronisation and threading policies. We note that current distributed object and component middleware systems support a fixed number of such synchronisation and threading primitives. OMG's CORBA, Microsoft's Component Object Model (COM) 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 only directly supports single threaded behaviour, but server programmers can use Java's threading primitives to construct multi-threaded behaviour on top of this.

As the synchronisation 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 synchronisation primitives and threading policies of COM and RMI. We subsequently define stereotypes for all the primitives that CORBA provides. These primitives can then be used during the design of applications based on other distributed object and component technologies too. Our approach therefore caters for design and property violation detection of all applications based on mainstream object and component middleware.

Recent advances in middleware technology have brought about component middleware technologies such as Enterprise Java Beans and the CORBA Component Model (CCM). Components representing business logic are hosted in the middleware's container. Component middleware technologies use existing object middlewares for establishing communication between components. For example EJB communication is achieved via RMI and CCM communication is done through CORBA. Therefore by providing semantics for the primitives of the underlying object middleware technologies we cater for the component middleware technologies as well.

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 define 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. We will specify the semantics of these stereotypes formally in Section 5.

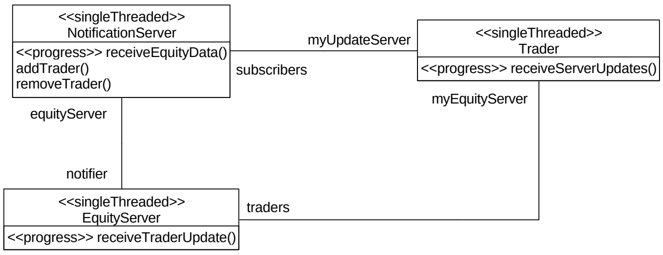

Fig. 2. Class diagram of Market Trading Scenario

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, Fig. 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 previously mentioned, this is the default threading policy in all 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 receiveServerUpdates() of Trader is invoked by an object of type NotificationServer to pass the EquityServer updates.

Synchronisation of remote operation invocations is a dynamic aspect and as such we define them in state diagrams. We use the synchronisation stereotypes mentioned above in those transitions of statecharts whose actions correspond to remote operation invocations. The statechart of the EquityServer in Fig. 3 initially starts in the idle state. After receiving a request for its exported receiveTraderUpdate method, 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 a request invocation upon the receiveEquityData method of the NotificationServer class in Fig. 2. Notice that the action name contains the name of the association-end used in the class diagram. From this information we can deduct that an EquityServer object requests a remote synchronous operation from a single-threaded NotificationServer server object. Finally, the EquityServer goes back to the idle state causing a reply to be sent back to the Trader instance who sent the updates. If a state diagram contains actions indicating receiving an operation request then the designer must also indicate the point at which a reply is sent back to the client object. An example of this is the receiveEquityData and receiveEquityData_reply actions in Fig. 3.

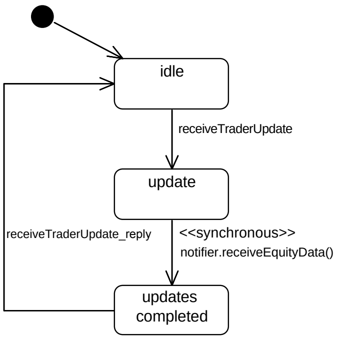

Fig. 3. EquityServer Statechart

Fig. 4 shows how the NotificationServer can register and unregister traders whilst in the idle state. Requests to be added or removed from the subscription list is replied to immediately via the addTrader_reply and removeTrader_reply methods respectively. 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.

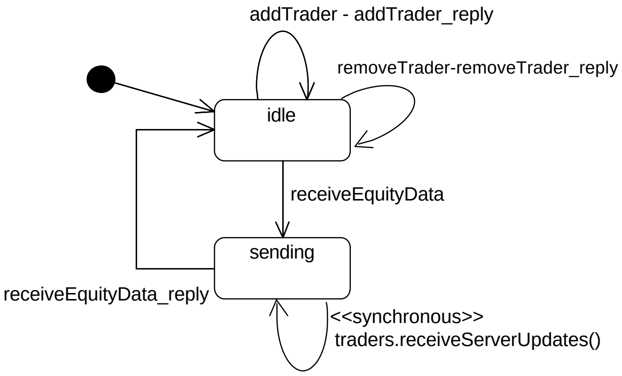
addTrader - addTrader_reply
removeTrader-removeTrader_reply
idle
receiveEquityData
sending
receiveEquityData_reply
<<synchronous>>
traders.receiveServerUpdates()

Fig. 4. NotificationServer Statechart

Fig. 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 invocations made to instances of type `EquityServer` are synchronous. After replying to the `receiveTraderUpdate` event the object returns to state `idle`.

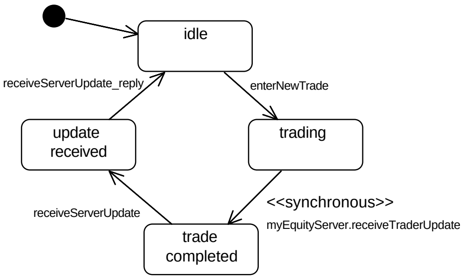

Fig. 5. Trader Statechart

Although class diagrams and state diagrams depict the static and dynamic characteristics of a system respectively, they both operate at a type-level of abstraction. In the case of distributed software architecture, it is often necessary to include instance-level designs as well. This is because the wide ranging dynamic behaviour of a distributed system depends on deployment configuration of an application and its environment. This also explains the need for component instantiation primitives in popular Architecture Description Languages such as Darwin [19].

Our initial solution for the inclusion of instance-level information was to derive it from the cardinality of association ends in class diagrams. This method was found to be infeasible for two main reasons. Firstly, the derived instance-level information depicts all potential instances of a class being connected to all other instances of its associated classes. We found that this is rarely the case in real applications. Secondly, cardinalities with an infinite nature such as one-to-many and many-to-many cannot be mapped to the category of finite-natured formal specifications that we would like to use. Moreover at run-time there will only exist a finite number of instances in a distributed system and by specifying this at design time one captures a more accurate description of the system.

Our revised solution replaces the class diagram cardinality information with UML object diagrams. These characterise component instances and their connectors in the deployed distributed software architecture. This approach addresses the two mentioned problems and has some additional advantages: UML object diagrams allow designers to model different run-time configurations of an application, which can be automatically verified against a given set of safety and liveness properties. Moreover, designers gain flexibility as they can verify different run-time configurations of an application without any modifications to the state or class diagrams. Deployment diagrams were not considered for this purpose as they force designers to indicate matters such as location and different types of resources, which are of no use to our approach and furthermore break some of the transparencies that middleware technologies aim to provide.

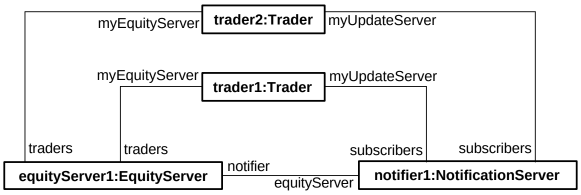

Fig. 6. Distributed Equity Trading System

Fig. 6 shows the deployment of the Distributed Equity Trading System Architecture using an object diagram. The run-time configuration of the application consists of two distributed `Trader` objects and one distributed `EquityServer` and `NotificationServer` object. The connectors are shown as links in the object diagram and reflect the association instances that exist between objects. Each link holds the names of the association-ends of its respective association. This is done to prevent any ambiguities in the case of having multiple associations between classes. In distributed object programs, these connectors would be implemented using distributed object references, which a client needs to request a remote operation execution.

In order to reconfigure the architecture to reflect, for example, that two equity markets work in conjunction with each other, we could reconfigure this architecture by sharing the same `NotificationServer` object but include a second `EquityServer` object. This would then be achieved by connecting `notifier1` in

Fig. 6 with the new `EquityServer` object (`equityServer2`) and having new `Trader` objects linked to the `equityServer2` instance.

4 Property Specification

Our prior work in the area of reasoning about distributed object architectures concentrated on the detection of potential deadlocks [17]. Deadlocks are a common source of errors in distributed applications. The absence of deadlocks is a necessary, but not sufficient criterion for the behavioural correctness of a distributed software architecture. Designers might want to specify more general safety and liveness properties. A *safety property* defines that no undesirable behaviour will be exhibited during the execution of a system, while a *liveness property* determines the desirable actions that will eventually be executed.

Unlike for absence of deadlocks, designers need to provide the assertions for safety and liveness properties, as they are specific to a particular distributed application. Traditionally notations such as Linear Temporal Logic [28] have been used to express these properties. The main drawback of this logic is the high level of expertise and fluency in formal notations that is required from a designer. Thus, such an option would break the formal specification transparency that our approach offers to designers. For these reasons we provide the designer with a technique of expressing desired properties in UML notation. There has been some research [23] carried out in order to create new categorisations of properties. However, for the purposes of our approach and the properties that we would like to offer the traditional safety and liveness classifications are sufficient and as we cannot benefit from different categorisation, we have based our work on well-understood conceptual foundations.

4.1 Safety Properties

We support the specification of safety properties for distributed object designs based on action orderings as these are more intuitive for the distributed system designer than the reachability of states. This is because the parallel execution of objects and components causes a large number of potential states many of which are implicit and not directly evident to designers. Moreover the notion of actions map nicely to operation invocations, which are the means of interaction in object middleware.

As was discussed above, the designer provides a UML state diagram for each object type in order to model the behaviour of instances of that type. Thus the order and occurrence of actions within a single UML state diagram govern the behaviour of individual instances. Using safety properties, we can determine whether the behaviour that is modelled locally in objects respects global correctness criteria.

We define safety properties by asserting global constraints on the order in which remote operation invocations may occur. We propose the use of state diagrams to determine these action ordering. We refer to these diagrams as *safety state diagrams* in order to distinguish them from those that determine object behaviour.

In order to define the intuitive meaning of these safety statecharts, we need to introduce the notion of traces and alphabets. A *trace* is a sequence of distributed operation invocations that is permitted by the state diagrams that govern the behaviour of individual distributed objects. A single element of a trace is an *action*, which we label with the name of the invoked operation. The set of operation names used in the union of all possible traces is called the *alphabet* of the distributed object system. Given a subset S of a distributed object system's alphabet A, we may *restrict* a trace of the system with S by deleting any actions that denote operations that are not in S.

The union of operation names that annotate transitions in a safety state diagram will be a subset of the system's alphabet. When checking for a safety property, we would like to ascertain that any trace of the distributed system restricted to the alphabet used in the safety state diagram is identical to the one described in the safety state diagram.

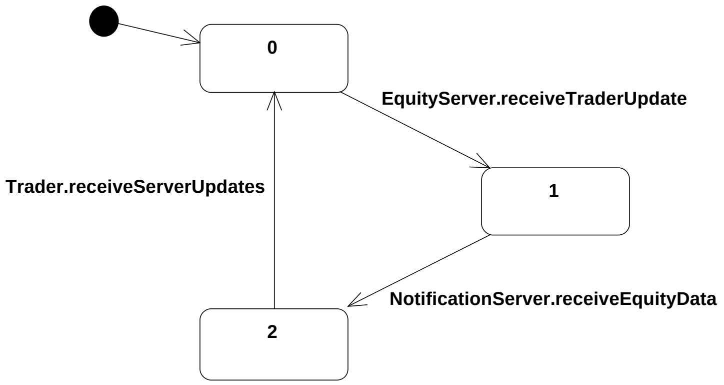
0
EquityServer.receiveTraderUpdate
Trader.receiveServerUpdates
1
NotificationServer.receiveEquityData
2

Fig. 7. Safety Property with Types

Fig. 7 shows a safety property for the scenario example discussed in Section 2. The purpose of the property is to ensure that trade activity is generated before any updates are sent to the traders. This property states that all possible execution traces of the application should respect the below recurring pattern in the given order:

1. An instance of the `EquityServer` class must receive a `receiveTraderUpdate` request
2. Next an instance of the `NotificationServer` class must receive a `receiveEquityData` request
3. Next an instance of the `Trader` class must receive a `receiveServerUpdates` request. Back to step 1.

The alphabet of the Trading system is the union of traces obtained from the interaction between all instances of the `EquityServer`, `NotificationServer` and `Trader` classes. In the case of the above safety property we have introduced a subset of the alphabet which needs to be matched by all traces obtained from any interaction. This safety alphabet consists of the actions shown in Fig. 7 enumerated over the instances of the corresponding types, obtained from the object diagram of Fig. 6.

In Fig. 7, the operation names are preceded by class names. The meaning of that construct is that we offer a non-deterministic choice of any objects that are instances of that class. There may be situations, however, where designers want to express safety properties for particular objects. Thus, we also support the enumeration of sets of object names from the object diagram that describes the distributed system architecture.

Fig. 8 demonstrates this by specialising the safety property in Fig. 7 by only including the `trader1` object. In general, designers simply wrap a comma separated list of objects in curly braces to indicate that the property targets certain instances instead of all instances associated with a class.

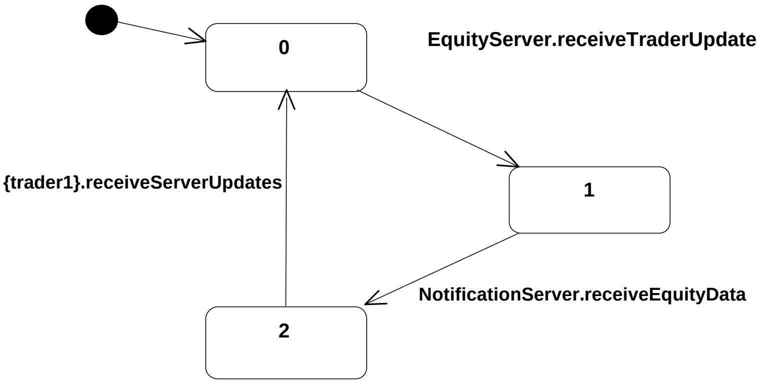
0
EquityServer.receiveTraderUpdate
{trader1}.receiveServerUpdates
1
NotificationServer.receiveEquityData
2

Fig. 8. Safety Property with Objects

By supporting the safety properties using action ordering we enable designers to express various higher level properties specific to a distributed application, such as mutual exclusion.

4.2 Liveness Properties

The focus of liveness properties is on the continuity of the execution of an application i.e. that a specific set of actions eventually happen. We currently support the progress property as defined by Magee and Kramer [20]. Intuitively, progress means that that it is always the case that an action from a given set will eventually be executed. Progress evaluates to the temporal logic property of "always eventually" Using progress properties, we can detect livelocks in application designs.

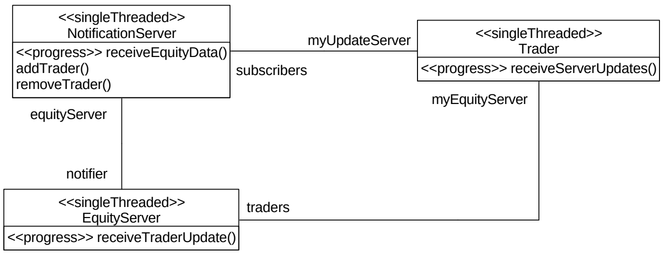

Fig. 9. Trading Order Liveness Property

We support the specification of progress properties by defining a ⟨⟨`progress`⟩⟩ stereotype that can be attached to operations of classes crucial to the progress of the application. Fig. 9 depicts an example that adds progress properties to the example scenario discussed in Section 2. In this case, we have identified that the respective methods of each class responsible for circulating new and update trade information are vital for the continuity of our application.

5 Formal Semantics of Stereotypes

Section 3 demonstrated our approach in producing annotated UML models of a distributed application. In order to make firm and accurate deductions about the dynamic behaviour of such an application, our approach prescribes a mapping of the design model into a formal specification. The formal specification is a description of when and in what way do parallel executing objects synchronise with one another. The point of synchronisation is found by analysing the state diagram of each object type. The method of synchronisation is derived by detecting the client-side synchronisation primitive and the server-side threading policy stereotypes specified for each interaction.

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 algebra that we have chosen for defining the semantics of the stereotypes are Finite State Processes [20] (FSP). We chose FSP because it is well-supported by a model checking tool.

In our approach we have derived FSP specifications for all combinations of client-side synchronisation primitives and server-side threading policies discussed in Section 3. By analysing the input annotated UML model we obtain the specific combination of synchronisation primitives and threading policies specified by the designer. By mapping each detected combination with its corresponding formal semantic and by finally composing all the formal specification fragments together, we obtain a formal specification of the overall application design.

There are two concepts that we commonly use across specifications of primitives and threading policies. Firstly we insert into the system specification, middleware-specific actions relating to the mediation of requests and results between a client and a server object. This is required for the correct modelling of the system's synchronisation behaviour. Secondly we directly support the notion of instance-level modelling in the formal domain by reflecting the specification from the object diagrams discussed in Section 3.

At a notation level there are two techniques that we use for the generation of FSP specifications, 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 synchronised/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 synchronised using the FSP relabelling mechanism. This Section discusses these specifications in detail.

5.1 Synchronisation 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 synchronisation primitive not only in CORBA, but also in RMI and COM. Fig. 10 shows the FSP specification for a *synchronous* call. A central component of any object middleware system is an Object Adapter(OA). The Object Adapter is the key entity in middleware technology, in terms of orchestrating the synchronisation between server objects and client requests. It is possible for an object adaptor to be responsible for handling more than one server object. However since there is no way of knowing this, our formalisation assumes the extreme case of appointing an object adapter for each server object. The role of an object adapter is directly mapped to the `OA` FSP process, which forms the synchronisation between `Client` and `Server` processes. 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 synchronise 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.

```
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}.
```

Fig. 10. Synchronous Stereotype Semantics

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. Fig. 11 shows the FSP specification for the *asynchronous* invocation method. Similarly to the previous case the `OA` process mediates the synchronisation of the `Client` and `Server` actions. However, in this case the client can engage in other actions infinitely often before it receives a call-back invocation from the server, via the OA. This is indicated by the "..." in the process, `otherExecutions`. This is made possible by the FSP choice operator "|", which introduces a non-deterministic method of executing alternate actions.

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.

5.2 Threading Policies

The primitives described in Section 5.1 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

```
Client=(sendRequest->OtherExecutions),
OtherExecutions=(...->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}.
```

Fig. 11. Asynchronous Stereotype Semantics

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. If the client is expecting a reply from its operation request it will receive a generated system-level exception. Fig. 12 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.

6 Formal Semantics of Properties

In order to automatically verify the generated formal specification of a system against user-provided properties, we need to translate the expressed safety and liveness properties into the process algebra domain. The property specifications need to be in the same notation as the system specification. In this section we discuss the generation and integration of property specifications expressed in Section 4.

```
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}.
```

Fig. 12. Semantics of Multi-Threaded Stereotype

6.1 Safety Property Semantics

In order to generate FSP to model the system at an object level of granularity we must refer to the object diagram. Fig. 13 shows the corresponding generated FSP process algebra for the safety property specified in Fig. 7. The class names in the transitions are replaced by a list of instance names obtained from the object diagram.

For example the server class name on the first transition is `EquityServer`. Consulting the object diagram shows that the list of instances of this class contains only one element, `equityServer1`. By further consulting the state and object diagrams we determine the list of client objects that are linked to and invoke operations from `equityServer1` – `trader1` and `trader2`. We can now construct the FSP action by combining the names of the clients, the server and the operation.

```
property SFY= ({trader1,trader2}.equityServer1.receivetraderupdate->S1),
S1=({equityServer1}.notifier1.receiveequitydata->S2),
S2=({notifier1}.trader1.receiveserverupdates->SFY
    |{notifier1}.trader2.receiveserverupdates->SFY).
```

Fig. 13. Safety Property Semantics Example

The above specification is composed of three sections, each section corresponding to each transition action of Fig. 7. As introduced in Section 4.1, the complete set of traces generated from the formal specification of the Trading scenario need to comply with the traces generated from the above safety property.

6.2 Liveness Property Semantics

Fig. 14 shows the generated FSP specification for the progress property example of Fig. 9. Similarly to the safety property example discussed in previous subsection, we make use of the object diagram to generate object-level specifications. Each annotated method is prefixed with the object names of the class type and further prefixed with the object name of instances linked to them in the object diagram. For example the progress property `EQUITYSERVER_PROGRESS0` addresses the source instance `equityServer1` as well as instances that can potentially invoke the operation `receiveTraderUpdate`, namely `trader1` and `trader2`.

7 Minimisation

The main challenge of verification of system properties using model checking techniques is the potential for state explosion [11]. There has been a growing trend of applying model checking techniques to more complex fields, such as software engineering, than its original field of use, hardware and protocol design. This growing complexity has turned this problem into a pivotal factor for

```
progress EQUITYSERVER_PROGRESS0=
  { trader1.equityServer1.receivetraderupdate,
    trader2.equityServer1.receivetraderupdate }

progress NOTIFICATIONSERVER_PROGRESS0 =
  { equityServer1.notifier1.receiveequitydata }

progress TRADER_PROGRESS0 =
  { notifier1.trader1.receiveserverupdates,
    notifier1.trader2.receiveserverupdates }
```

Fig. 14. Progress Property Semantics Example

deploying finite-state verification techniques. Attempting to verify distributed object systems amplifies this problem. This is due to the high degree of autonomy present between objects executing in parallel, giving way to a very large number of possible execution traces. As a consequence, the model's state space grows exponentially with respect to the number of objects involved, rendering naive brute force approaches unusable.

We tackle the state space explosion problem from a number of different angles. Our work concentrates on exploiting middleware characteristics for state reduction and the generated process algebra only takes into account a small finite number of synchronisation primitives and threading policies.

The insight of knowing our problem domain is further reflected in the underlying process algebra specification that we generate. The behaviour of each distributed object is described in one FSP process. However, only the actions that deal with making or receiving remote method requests, as described in the UML state diagram, are exposed. The execution of local method calls, the interaction between a possibly large number of local objects, as well as the operation parameter and return values have no implications on the emergent synchronisation behaviour of a distributed application and can therefore be ignored. Abstracting from these details reduces the state space significantly. We can achieve further reductions by considering the way in which middleware implements distributed interactions.

In all object and component-oriented middleware systems there is a middleware component that is responsible for receiving all incoming requests for a server objects and for delivering them to the appropriate object implementation for servicing. In CORBA, this component is called the Object Adapter, COM provides a Service Control Module that has this function and Java/RMI uses the activation interfaces that are contained in the RMI daemon. We subsume these components under the notion of *object adapters* below.

An object adapter decouples client from server objects. All operation invocation requests are initially received by the object adapter on the server object's hosts and the adapter then forwards them to the server objects as exemplified in Fig. 15. Likewise any reply of the server object will be transmitted via the object adapter. Since the object adapter has a fixed interface of only two actions – for receiving and replying to requests – and the client objects can only interact with server objects through these two actions, we can achieve further minimisation of the state space: In a scenario of n clients invoking m different server methods, we can reduce the combination of interactions from $n \times m$ to $n \times 2$. This means that the final state space will be independent of the number of methods that a server object type exports. As the final state space is the product of the size of the component states during parallel composition, this reduction will greatly reduce the final state space.

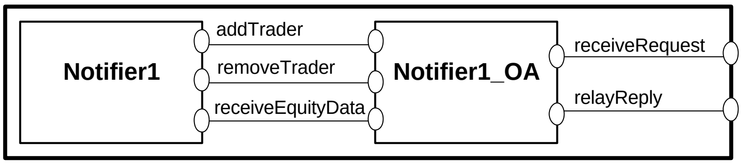

Fig. 15. Middleware Aware Minimisation

8 Model Checking

Model checking provides a means of automatically verifying input process algebra for a given set of properties. This is achieved by building a state space representation of the specification and exhaustively searching this space to ensure that all states are valid with respect to the desired properties. Model checking tools vary in features such as the data structures they use to hold the state space and the algorithm they use for searching the state space. Whilst such features may affect the performance of the model checking performance by a given factor, they are quite similar in the way they approach the problem.

The FSP process algebra is provided with the Labelled Transition System Analyser(LTSA) model checker. The LTSA model checker generates a Labelled Transition System(LTS) for each of the generated FSP processes and applies our minimisation methods. These LTSs are then composed together into one large LTS, taking into account the required synchronisation between the objects as specified in the FSP processes. This final LTS represents the state space of the application model. Subsequently the LTSA carries out an exhaustive search of the state space for verification purposes. It is the exhaustive nature of the search that gives formal verification methods their rigorous powers and high reliability in finding the most subtle of errors. In case of a property violation detected the LTSA outputs the shortest trace of actions that causing the violation.

A deadlock situation is detected when a state with no outgoing transition is found. This indicates that there is no further states that the modelled application can enter, causing the system to halt and deadlock. Fig. 16 shows the trace of actions leading to a potential deadlock in the Trading scenario we have been discussing. The trace shows how initially `trader1` sends the results of an equity transaction to the `equitysever1` instance. The instance `equityserver1` receives this and successfully requests the object `notifier1` to send equity price updates to the traders. The deadlock occurs when `trader1` again sends new transaction information to `equityserver1`, but `notifier1` immediately follows this up by sending another update to `trader1`. At this stage both `trader1` and `notifier1` are blocked and any further synchronous invocations to these two objects would block the caller for ever. Thus when this does happen the system enter a deadlock status.

```
Trace to DEADLOCK:
 trader1.equityServer1.receivetraderupdate
 equityServer1.receivetraderupdate
 equityServer1.notifier1.receiveequitydata
 notifier1.receiveequitydata
 notifier1.trader1.receiveserverupdate
 trader1.receiveserverupdate
 trader1.receiveserverupdates_reply
 notifier1.receiveequitydata_reply
 equityServer1.receivetraderupdate_reply
 trader1.equityServer1.receivetraderupdate
 equityServer1.receivetraderupdate
 notifier1.trader1.receiveserverupdate
 equityServer1.notifier1.receiveequitydata
 trader2.equityServer1.receivetraderupdate
```

Fig. 16. LTSA Deadlock Trace

A safety property violations is detected when the model checker finds a trace of actions containing one or more actions of the safety property, where the safety property's action ordering is not followed. Fig. 17 shows an example of the safety property violation depicted in Fig. 8. This safety property stated that out of all active traders only the instance, `trader1` should be informed of new equity updates via instances of the `NotificationServer` class. The property is violated in our distributed object model as the `NotificationServer` object might send notifications to both trader objects (refer to Fig. 6). In this case an update was sent to the instance `trader2`.

Progress violations are detected by looking for any set of actions that form an infinite cycle in which one or more of the progress actions are not included. Such set of actions are referred to as a terminal set. The LTSA reports this violation by showing a trace of actions to the terminal set and the terminal set itself.

```
Trace to property violation in SFY:
 trader1.equityServer1.receivetraderupdate
 equityServer1.receivetraderupdate
 equityServer1.receivetraderupdate_reply
 trader1.equityServer1.receiveInvocationReply
 equityServer1.notifier1.receiveequitydata
 notifier1.receiveequitydata
 notifier1.trader2.receiveserverupdates
```

Fig. 17. LTSA Safety Violation Trace

9 Relating Results

A key requirement of this research is to enable designers to reason about distributed object designs entirely using the UML notation. To attain this goal, we translate the traces of actions generated due to safety and liveness property violations into UML Sequence Diagrams. Sequence Diagrams offer a comprehensive and intuitive manner of showing a designer counter-examples of how their properties can be potentially violated.

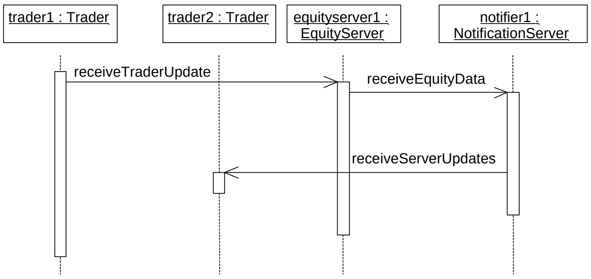

Fig. 18. Violation of Safety Property depicted in Fig. 8

Fig. 18 shows the sequence diagram generated for the safety property violation shown in Fig. 17. The sequence diagram shows how the application design allows a potential execution where by the instance `notifier1` could send updates to the `trader2` object, thus violating the safety property.

We envisage the process of design verification to be iterative. So at this stage the user should make any required modifications, to make their design rid of the potential problem and repeat the verification on the new modified design model.

10 Evaluation

In order to analyse the effectiveness of the suggested minimisation methods we have carried out an evaluation using the equity trading scenario introduced in

Section 2. The experiment is based on the configuration shown in Fig. 6 and carried out on a x86 architecture machine with dual 1.7GHz Xeon processors and 1GB of memory. The variant was the number of `Trader` instances executing in parallel along with the instances, `notifier1` and `equityServer1`. The main point of interest was the size of the state space gained by using the different approaches.

Fig. 19 shows the results of the evaluation. Not shown in the chart is the size of the maximum state space which ranged from 2^{31}-2^{71} for the set of traders shown on the x-axis. The line on the left hand side in the chart plots the state space gained by using the Compositional Reachability Analysis (CRA) [1] of the LTSA model checker without applying any minimisation. The line on the right hand side shows the performance of our minimisation technique. The CRA line is discontinued for all values above 7 traders since the model checker runs out of memory after generating 5300000 states. Whilst both techniques exhibit exponential growth, our minimisation approach has a lower growth factor and supports the validation of larger systems. In this case our minimisation approach was able to almost double the performance of the CRA method. Moreover our early evaluation was carried out on a relatively modest machine. We envisage designers to operate the MUDV tool on a stronger machine, thus yielding even better results.

When interpreting absolute state space size, the reader should bear in mind that in realistic distributed applications the total number of distributed objects is fairly low. The distributed object architecture that we discussed in [4], for example, deployed 10 distributed objects for the trading system integration of the sixth largest German bank. The reason for this small number of objects is that while application may be composed of a large number of objects and components, developers typically choose to only make a small portion of them available for distributed interactions and the rest execute locally or are deactivated. This is to minimise the resources required by the distributed application which include network bandwidth and memory for holding the stubs and skeletons of distributed objects.

11 MUDV Tool

Our “Modelchecking UML Design Verifier (MUDV)” is the tool that we have built in order to apply and evaluate our approach. The core task of the MUDV tool is to generate process algebra specification from input annotated UML models. An overview architecture of the tool is shown in Fig. 20. Designers use off-the-shelf UML CASE tools to create their system design. Most of these tools now support export into the Object Management Group’s XML Metadata Interchange [27] (XMI) format. XMI is a standard for encoding UML models in XML. The wide support of XMI in UML CASE tools and the intuitive methods of information extraction from XML documents makes it a suitable input notation for the MUDV.

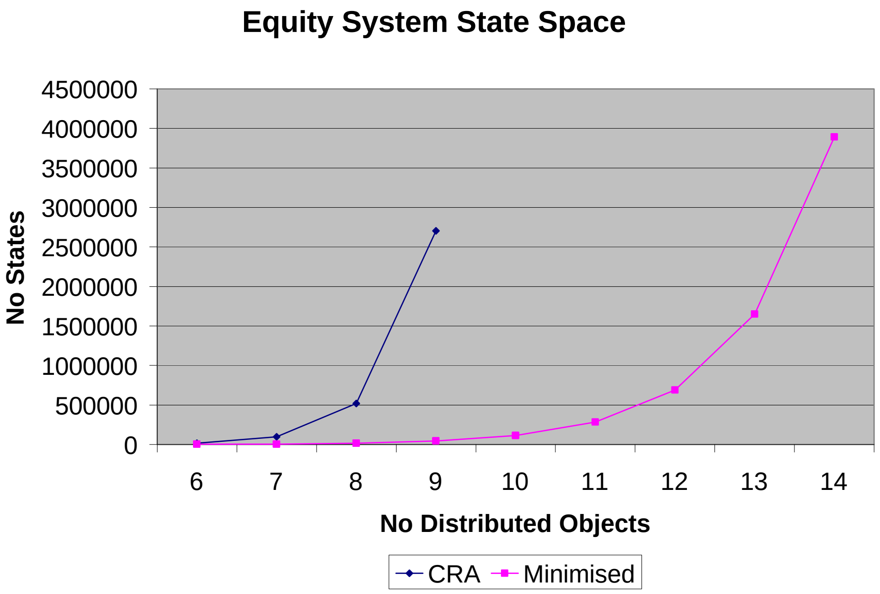

Fig. 19. State Space Size Comparison

The MUDV tool generates the process algebra specification of the input design model. We then use the Labelled Transition System Analyser (LTSA), a model checker for FSP, to verify the conformance of all possible execution traces with the provided properties. This architecture gives designers the flexibility of using any UML design application with XMI support as well as decoupling it from the model checking tool. This allows us to create mappings to different formal specifications and integrate them into the overall architecture seamlessly.

Fig. 21 shows the design of the MUDV tool. The main feature of this design is the use of the Visitor pattern [7], which accommodates the seamless integration of new formal specification mappings. The three UML diagram types that we use in our approach are represented by the three classes which realise the general MUDVElement interface. All classes of type MUDVElement support the method `accept` which takes as input-parameter a reference to an instance of the general type Visitor. Instances of this type hold the functionality for producing specific types of formal specification. Once a MUDVElement has been passed a Visitor instance, via the MUDVTool class, it invokes the appropriate method for it to be analysed and mapped to a specification. This design makes the MUDV tool flexible and with low cohesion between its components. We have implemented the plug-in for the generation of FSP specifications and are currently creating a SPIN [13] plug-in to demonstrate the general applicability of our approach.

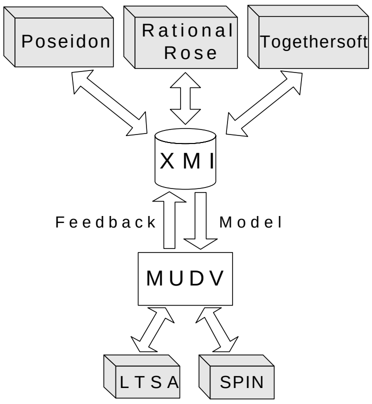

Fig. 20. MUDV Architecture

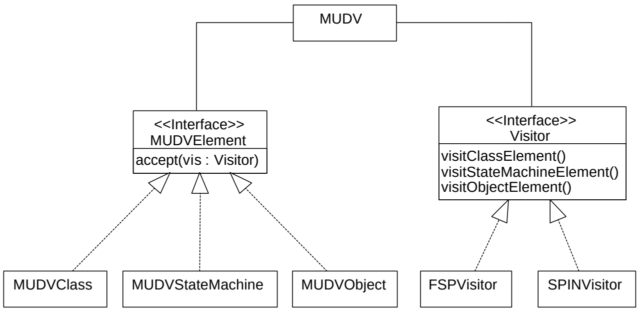

Fig. 21. MUDV Core Design

12 Related Work

The notion of accelerating the development life-cycle of software methodologies through automation is an appealing idea with a long history. This originally started with the introduction of Computer Aided Software Engineering(CASE) tools in the 80s. The OMG's Model Driven Architecture [6] is the most recent attempt at standardising the automation of deployment design and substantial implementation tasks.

The work done in [18] 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 separate process. This is not the case for object middleware as many server objects can run in the same process. In [15] automatic deadlock free synthesis of COM/DCOM architecture connectors is achieved from the dynamic behaviour specification of the components, but no general safety or liveness properties are enforced. The work of the same authors reported in [14] is related in that the authors also translate UML designs into SPIN models. The most important difference is that we explicitly use stereotypes to express the synchronisation and threading policies and that we aim to hide the complexity of using a model checker completely.

In [30], 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.

The pUML [5] research attempts to give formal semantics of UML diagrams using the Z notation, allowing them to verify UML models. The approach taken in the Hydra project [21] is provide a mapping from the UML metamodel to formal language metamodels. This mapping leads to a set of rules which govern the automation of a particular formal specification. This mapping does not cover the whole of the UML diagrams set, specifically UML stereotypes which is the basis of conveying middleware related information in our approach.

13 Summary and Conclusion

It is our belief that the advances in distributed object and component technologies need to be complemented by new software engineering methods and tools to guide developers in increasingly complex situations [3]. The work presented in this paper focuses on automatically verifying the non-deterministic synchronisation behaviour of object middleware applications, caused by the interaction between distributed objects executing in parallel.

This paper reports on a number of new contributions. By extending the semantics of UML state diagrams and the introduction of new stereotypes, we provide designers with the ability to express safety and liveness properties in the UML notation. Formal specifications of the properties are automatically generated and composed with the process algebra specification of the system. Feedback on any property violations is done via UML sequence diagrams, maintaining transparency of the heavy formal specification to designers. Traditional property specification techniques, such as those using temporal logic, offer a more expressive power than our approach. However, this comes at a cost of being user-unfriendly and difficult to master, which we have often found to be a stumbling block for the industrial adoption of these techniques. Taking into consideration the fact that this research is aimed at supporting general industrial practitioners with little or no experience of formal techniques, we feel that our approach maintains a suitable balance in this trade off.

Our second contribution is the integration of models for the deployment of distributed components and their interconnection via the use of UML object diagrams. This enables designers to experiment with and verify different run-time configurations of a distributed object system without any modification required to other models. The analysis of an application at an instance level is further reflected in the generated process algebra and the feedback sequence diagrams.

Furthermore, by solely modelling object interactions where indicated in an object diagram we reduce the complexity, and thus the state space, of the formal model, leading to more efficient model checking.

We also presented and evaluated the methods we employ to tackle the state explosion problem. By exploiting domain specific (object middleware) knowledge of the nature of the applications being modelled, we build further minimisation methods on top of what is typically offered by model checkers, with the goal of reducing a model's state space. Only actions that correspond to remote object interactions, making up the synchronisation behaviour of an application, are model checked. By modelling the entity responsible for receiving and delivering requests in a distributed system we also gain incremental minimisation.

Even though the minimisation techniques we presented above greatly improve the usability of our approach and facilitate verification of medium-sized industrial models, we are aware that they do not yet scale up to large scale distributed systems with several hundred distributed objects. In order to achieve this scalability, we need to take advantage of the fact that these objects are often isolated from each other and partitioned into federations of distributed objects. Fortunately, the federations of objects that do interact with each other are rarely larger than the ones that we can model check. We can then analyse these federations in isolation from each other and in that way achieve the scalability required in practice.

We are currently developing a new semantic mapping of UML models into Promela specifications, the input notation for the SPIN model checker. This will demonstrate the general applicability of our approach to various formal semantics as well as benefiting from the advantages of the SPIN model checker, such as support for timeouts, assertions and optional compact searches as opposed to an exhaustive one. We plan to further evaluate our approach by carrying out a case-study obtained from our industrial collaborators.

The techniques that we have outlined in this paper are providing feedback on *qualitative* properties of distributed object design. We have started investigating reasoning techniques for *quantitative* properties, such as scalability, performance and reliability of distributed object and component designs. It would be highly desirable to avoid costly risk mitigation iterations during a development process and address the question of whether an architecture scales and performs efficiently and reliably by analytic means. The performance modelling literature includes a large body of work on *stochastic process algebras*, which use distribution functions with which transitions are executed [12, 9, 8]. It seems natural to extend the research that we presented here to performance, scalability and reliability properties of UML models that can then be expressed and analysed with stochastic process algebras.

In [24, 25], we have described xlinkit, a consistency checker that can be used to validate the static consistency of software engineering documents represented in XML. That research is largely complementary to the techniques for establishing behavioural consistency that we have presented in this paper. A combination of the two approaches would enable us to statically validate the correctness of the various relationships between the different diagrams, such as that for each object in an object diagram, there is a class in a class diagram whose name is identical to the type of the object and would therefore enhance the usability of our our model checker. We therefore plan to address this integration in the immediate future.

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. W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, April 2000.
3. W. Emmerich. Distributed Component Technologies and their Software Engineering Implications. In *Proc. of the* 24^{th} *Int. Conf. on Software Engineering, Orlando, Florida*. ACM Press, 2002. To appear.
4. W. Emmerich, E. Ellmer, and H. Fieglein. TIGRA – An Architectural Style for Enterprise Application Integration. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering, Toronto, Canada*, pages 567–576. IEEE Computer Society Press, 2001.
5. A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. In Pierre-Alain Muller and Jean Bézivin, editors, *Proc. of the International Conference on the Unified Modeling Language (UML): Beyond the Notation*, volume 1618 of *Lecture Notes in Computer Science*, pages 336–348. Springer-Verlag, 1998.
6. D. Frankel. *Model Driven Architecture – Applying MDA to Enterprise Computing*. OMG Press. Wiley, 2003.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.
8. S. Gilmore, J. Hillston, and M. Ribaudo. An efficient algorithm for aggregating PEPA models. *IEEE Transactions on Software Engineering*, 27(5):449–464, 2001.
9. N. Götz, U. Herzog, and M. Rettelbach. The Integration of Functional Specification and Performance Analysis using Stochastic Process Algebras. In *Proc. of the* 16^{th} *Int. Symposium on Computer Performance Modelling, Measurement and Evaluation (PERFORMANCE 93)*, volume 729, pages 121–146. Springer, 1993.
10. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.
11. David Harel, Orna Kupferman, and Moshe Y. Vardi. On the complexity of verifying concurrent transition systems. In *International Conference on Concurrency Theory*, pages 258–272, 1997.
12. J. A. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, Dept. of Computer Science, University of Edinburgh, UK, 1994.
13. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
14. P. Inverardi, H. Muccini, and P. Pelliccione. Automated Check of Architectural Models Consistency using SPIN. In *Proc. of the* 16^{th} *Automated Software Engineering Conference, Coronado Island, CA*, pages 346–349. IEEE Computer Society Press, 2001.
15. P. Inverardi and S. Scriboni. Connector Synthesis for Deadlock-Free Component Based Architectures. In *Proc. of the* 16^{th} *Automated Software Engineering Conference, Coronado Island, CA*, pages 174–181. IEEE Computer Society Press, 2001.

16. N. Kaveh. Model Checking Distributed Objects. In W. Emmerich and S. Tai, editors, *Proc. of the* 2^{nd} *Int. Workshop on Distributed Objects, Davis, Cal, Nov. 2000*, volume 1999 of *Lecture Notes in Computer Science*, pages 116–128. Springer, 2001.
17. N. Kaveh and W. Emmerich. Deadlock Detection in Distributed Object Systems. In V. Gruhn, editor, *Joint Proc. of the* 8^{th} *European Software Engineering Conference and the* 9^{th} *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 44–51. ACM Press, 2001.
18. J. Lilius and I. Paltor. A Tool for verifying UML models. In *Proc. of the* 14^{th} *Int. Conference on Automated Software Engineering, Cocoa Beach, Florida*, pages 255–258. IEEE Computer Society Press, 1999.
19. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.
20. J. Magee and J. Kramer. *Concurrency: Models and Programs – From Finite State Models to Java Programs*. John Wiley, 1999.
21. W. E. McUmber and B. H. C. Cheung. A General Framework for Formalizing UML with Formal Languages. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering, Toronto, Canada*, pages 433–442. IEEE Computer Society Press, 2001.
22. R. Monson-Haefel. *Enterprise Javabeans*. O'Reilly UK, 1999.
23. G. Naumovich and L. A. Clarke. Classifying Properties: An Alternative to the Safety-Liveness Classification. Technical Report UM-CS-2000-012, Dept. of Computer Science, University of Massachusetts in Amherst, 2000.
24. C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: A Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2002. To appear.
25. C. Nentwich, W. Emmerich, and A. Finkelstein. Static Consistency Checking for Distributed Specifications. In *Proc. of the* 16^{th} *Automated Software Engineering Conference, Coronado Island, CA*, pages 115–124. IEEE Computer Society, 2001.
26. Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.3*. 492 Old Connecticut Path, Framingham, MA 01701, USA, December 1998.
27. 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, October 1998.
28. A. Pnueli. The Temporal Logic of Programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, Providence, R.I., 1977.
29. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
30. Sebastian Uchitel and Jeff Kramer. A Workbench for Synthesising Behaviour Models from Scenarios. In *Proc. of the* 23^{rd} *Int. Conf. on Software Engineering, Toronto, Canada*, pages 188–197. ACM Press, 2001.