Translation from Adapted UML to Promela for CORBA-based Applications

J. Chen and H. Cui

School of Computer Science, University of Windsor
Windsor, Ont. Canada N9B 3P4
{xjchen,cui}@uwindsor.ca

Abstract. Nowadays, many distributed applications take advantage of the transparent distributed object systems provided by CORBA middlewares. While greatly reduce the design and coding effort, the distributed object systems may also introduce subtle faults into the applications, which on the other hand, complicate the validation of the applications. In this paper, we present our work on applying SPIN to check the correctness of the designs of CORBA-based applications, taking into account those characteristics of CORBA that are essential to the correctness of the applications. In doing so, we provide adaptations to UML, so that the CORBA-based applications can be modeled with succinct yet sufficient details of the underlying middlewares. An automated translation tool is developed to generate Promela models from such UML design models. The translation tool embeds the behavioral details of the middleware automatically. In this way, the software developers can stay in their comfort zone while design faults, including those caused by the underlying distributed object systems can be pinpointed through the verification or the simulation with SPIN.

Keywords: Distributed Object Systems, Middleware, Formal Specification and Verification, Model Checking, UML, SPIN, CORBA.

1 Introduction

Nowadays, many distributed applications take advantage of middlewares. A middleware encapsulates the heterogeneity of the distributed computing environment and provides a consistent logic communication media for distributed applications. This greatly reduces the design and coding effort. Many general object-oriented distributed applications choose distributed object system (DOS) middlewares, where objects are made available by the middleware beyond process boundaries. To promote the interoperability of DOS middlewares, a widely accepted standard, CORBA, is established by OMG. Today, most DOS middlewares are either CORBA-compatible or similar to CORBA on architecture level.

However, distributed object systems may also introduce subtle faults into the applications. This is because application designers usually tend to overlook

the differences between a local object and a remote object made transparently available. For example, phenomena like the serialization or blocking of remote object invocations may occur due to various reasons such as the resource configuration of the remote objects or the synchronization mode of the invocation itself. Overlooking such phenomena may under certain circumstances affect the business logic of the applications.

In this paper, we present our work on applying SPIN [6, 7] to check the correctness of the designs of CORBA-based applications, taking into account those characteristics of CORBA that are essential to the correctness of the applications.

Formal verification techniques, such as SPIN model checker, rely on formal specifications of the design documents. However, it is hard for ordinary software developers to grasp formal specification languages. One of the possible solutions is to define an automated translation from graphical, easy-to-use design models into the formal specification languages. For object-oriented applications, UML has been recognized as an industrial standard and is commonly used for software design. Here we assume that the design specifications of CORBA-base distributed applications are given in UML notations. As the UML notations stretch over almost all aspects of software artifacts, translating the entire domain of UML notations is not feasible. We choose to adapt and translate an essential subset of UML class diagram, statechart diagram and deployment diagram, which provide sufficient information about the dynamic behavior of the applications.

When building a verification model for a CORBA-based application, the behavior of a remote object should not be over-simplified as that of a local object. On the other hand, as the verification process is always resource intensive, the verification model must be succinct and should not accommodate a *complete* middleware model. In this paper, we construct an abstracted CORBA middleware model in Promela, considering those and only those characteristics of CORBA that are essential to the correctness of the applications, namely: (i) *binding*: the connection of a client to a remote object; (ii) *thread policy*: the POA (Portable Object Adaptor) policy which governs the service of remote objects; (iii) *synchronization mode*: the mode for client-side remote method invocations.

Accordingly, we made a few adaptations to UML notations, including (i) stereotypes for classes, methods and interfaces; (ii) pre-defined classes for client binding, remote method invocations and thread policies; and (iii) pre-defined component types for describing CORBA facilities (POAs and ORBs). These adaptations serve as the interface of the abstract CORBA middleware model. In this way, we preserve the clarity of the design models: The CORBA middleware remains a black-box in the design models. While generating the corresponding Promela model from such design models, our CORBA-middleware model is integrated automatically.

We have developed a translation tool called CUP, which generates the corresponding Promela source code from adapted UML diagrams. A concrete system is created from a UML deployment diagram using the elements defined in

UML class diagrams. The behavior of the elements is defined in UML statechart diagrams. The statechart diagrams in our design model are built upon parameterized method calls, change events, guard conditions and value assignment actions. Currently, CUP takes as its input only specially formatted XML file that represents the UML diagrams. The files are actually simplified standard XMI representation of the UML diagrams. We plan to use the XMI representations as the input of CUP in the future.

To reduce verification complexity, we translate most of the methods into Promela *inlines*, which is the most efficient way to simulate method calls in Promela [6]. For this purpose, we require that the behavior of a class is defined on per-method basis. Consequently, we eliminate all concurrency elements in a statechart diagram (sync, fork, join, concurrent regions in composite states). Instead, the concurrency is achieved through some specially stereotyped methods, which are specified as *running on its own execution thread*.

To further reduce the complexity, we define specially named composite states in statechart diagrams to identify atomically executed blocks. Thus, transitions dealing with only local variables can be organized into atomic execution blocks which eliminates some interleavings that do not affect the truthfulness of the correctness properties.

We assume that readers have basic knowledge of Promela/SPIN and UML. In Section 2, we give a brief introduction to parts of CORBA relevant to our work. A motivating example is described in Section 3. This example is used through out the paper to demonstrate the UML adaptation, translation and the verification results. Our adapted UML design model for CORBA-based applications is introduced in Section 4. In Section 5, we explain the generation of Promela code from the design model, especially the realization of the distributed object systems. Conclusion, related work and some final remarks are given at the end.

2 A Brief Introduction to CORBA

CORBA is a widely accepted standard for DOS middlewares established by OMG. A CORBA middleware makes objects in a distributed application accessible beyond process boundaries. The core component of the middleware is the object request brokers (ORB), which is responsible for finding a proper remote object to service a client (binding), communicating the method invocation requests to the remote objects and transferring the result back to the clients when it becomes available. An ORB may be *single-threaded* or *multi-threaded*. To be remotely accessible, a remote object must be managed by a POA object and registered to the naming space. The POA is the interface between the objects and the ORB. A POA manages remote objects, interprets the invocation requests made to the objects and activates proper servant threads to service the requests. How a POA performs is determined by its *policies*. The policy we are interested in is the *thread policy*, which specifies the thread model used for the servant threads:

- *main-thread*. All POAs in an ORB with this thread policy will share a servant thread provided by the ORB for all method invocations.
- *thread-per-POA*. The POA will create one servant thread to handle all method invocations.
- *thread-pool(n)*. The POA will create a set of n servant threads to handle method invocations.
- *thread-per-object*. The POA will create a servant thread for each remote object it manages.
- *thread-per-client*. The POA will create a servant thread for method invocations from each client of an object it manages.
- *thread-per-request*. The POA will create a servant thread for each method invocation request.

POA thread models can be used in either *single-threaded* or *multi-threaded* ORB. However, in a *single-threaded* ORB, the performance of any POA will be the same as that of a *main-thread* one.

Since method invocations may share a thread in ORB or POA according to the thread policies, internal blocking is introduced to the application, which may cause deadlocks or request re-ordering.

To find a remote object for service, a client must identify to the ORB, the remote object(s) it wishes to connect to and the ORB will locate the proper remote object for it accordingly. This process is called *binding*. Common binding options are:

- Request to bind to a specific remote object. The object is identified by its unique identifier.
- Request to bind to a remote object of a certain type. The type is identified by the remote interface of the object.
- Request to bind to a remote object of a certain type in a certain POA. The POA is identified by its unique identifier.

A remote method may be invoked in different synchronization modes. Typically, the invocation is *synchronous*: the client blocks until the remote method call returns. When a method requires no return values, the invocation can be made *asynchronously*: the client will continue its execution as soon as the request is received on the other end, without waiting for its completion. If the method invocation involves lengthy computation, *deferred synchronous* call is useful. In this invocation mode, the client continues execution after issuing the request, and *pulls* the result of the remote method call sometime later.

In the next section, we give a motivating example to show the design of distributed applications with CORBA middleware and the possible faults that may be introduced.

3 A Motivating Example

In an *auction* system, multiple clients (bidders) can compete for items. A bidder can withdraw from bidding at any time, and the bidding of an item terminates

when all bidders have withdrawn from the competition for that item, or when the pre-set price limit for that item is reached. When the bidding of an item terminates, whoever offered the highest bid first will succeed in the bidding. Suppose for fault-tolerance purpose, there are two servers running concurrently to service the bidders. Each of them maintains a local copy of the bidding status. A bidder is allowed to connect to any one of the servers, yet for fairness reason, only one connection is allowed for each bidder. The auction server will refuse a connection if another connection from the same bidder (determined by its identifier) already exists.

Let us assume that such a system is designed using CORBA middleware in the following way: Two remote objects are used as auction servers. The bidding status is represented as replicated local data of these objects. The two auction servers synchronize with each other through remote method invocations to keep the bidding status consistent. When two servers try to update the bidding status simultaneously, a token is used to determine which server should update first. An active object is used to represent a bidder. Each bidder object connects to an auction server through a client that is bound randomly to an auction server. To make a bidding request, the client invokes a remote method on the auction server it is connected to.

Below are some of the properties of the applications we are interested in and that can be verified with model checking tools:

- The design should be deadlock-free.
- The bidding process should eventually complete.
- One and only one of the bidders will eventually succeed in bidding an item.
- The successful bidder must hold a bid higher than or equal to the other bidders.
- The successful bid should be no higher than the pre-set price limit.

The correctness of such properties may vary depending on whether we model the middleware behavior or simply treat the remote object invocations similar as local calls. For example, an application according to the above design runs well under *multi-threaded* ORBs and *thread-per-client* POAs but it may run into deadlock if we use *thread-per-object* POAs. In the latter case, suppose bidder b_1 made a bidding request to auction server s_1 and bidder b_2 made a bidding request to auction server s_2 simultaneously. Server s_1 tries to synchronize with server s_2 on behalf of b_1: it makes a remote method invocation on auction server s_2 for the update of the bidding status, while vice versa, server s_2 tries to synchronize with server s_1. However, both these method invocations are blocked because the only servant threads on server object s_1 and s_2 are already occupied by bidders b_1 and b_2 respectively when they made the method invocations on their respect server object. As b_1 and b_2 will not release the servant threads of s_1 and s_2 respectively while they wait for the results of their bidding requests, the execution runs into a deadlock state. Such a deadlock situation cannot be identified if we model remote method invocations in the same way as local ones.

4 Adapted UML Diagrams

In this section, we introduce our UML design model for CORBA-based applications as the input of the CUP tool.

The design model consists of a set of self-executable active objects, ORB and POA components and remote object components deployed into the POA components. Each object and component is assigned a unique and publicly known integer ID. We assume all objects and components are persistent. We do not consider the creation, de-activation and destruction of any active objects, remote object components, POA or ORB components.

Correspondingly, the input model of the CUP tool consists of a UML deployment diagram, which creates and deploys all object instances in the application, UML class diagrams and the UML state chart diagrams, which are for the specification of the objects. Each method defined in the class diagram is associated with a statechart diagram to specify its behavior. The behavior of a class is specified by the collection of its method statechart diagrams.

In the following, we introduce the syntax and the semantics of the adapted deployment diagrams, class diagrams and statechart diagrams.

4.1 The Deployment Diagram

The deployment diagram specifies the following aspects of the application:

- The object and component instances and their unique identifiers;
- The *deployment relationship* between remote objects and POAs, between POAs and ORBs, is used to specify the management relationship among them.
- The thread policies for POAs and ORBs are specified by deploying a *Policy* object to a POA or an ORB.

The deployment diagram contains two nodes: a CORBA node and a *Clients* node.

The CORBA node forms the server-side of the application. It contains one or more ORB components. An ORB component contains a root POA component and possibly a *Policy* object. The latter defines the thread policy of the ORB. If omitted, the default thread policy is *single-threaded*. The root POA component contains zero or more remote object components and zero or more POA components. Each POA component also contains zero or more remote object components and zero or more POA components. A remote object is expressed as a component which implements a remote object interface. A POA component may contain a *Policy* object, which defines the thread policy of the POA. The thread policy of root POA is always *main-thread*. A POA without a *Policy* object inherits the thread policy of its parent POA.

The *Clients* node contains a set of *Process* components. They form the client-side of the application. Each *Process* component contains an active object, which is self-executable and serves as the starting point of the execution.

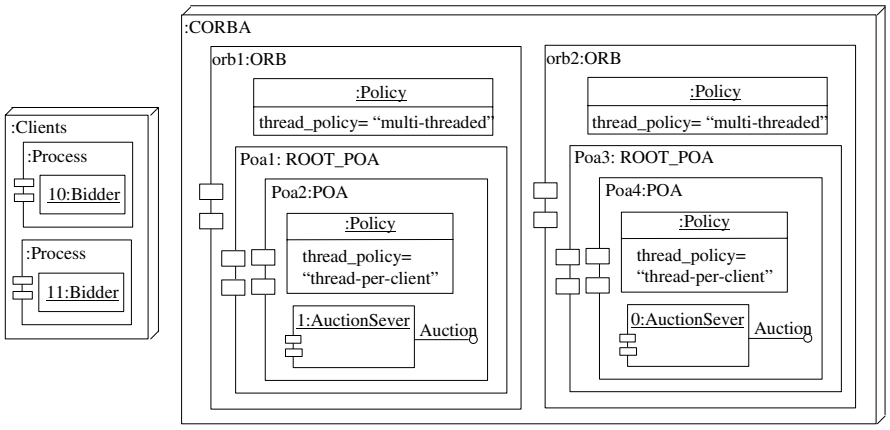

Fig. 1. Deployment diagram of the online auction example

Figure 1 shows the deployment diagram of the auction example, in which two bidders participate in the auction of an item. Here, the CORBA node contains two ORB components *orb1* and *orb2*, both specified as *multi-threaded*. In the root POA of *orb1*, we have POA component *Poa2* whose thread policy is *thread-per-client*. With this policy, it manages a remote object of class *AuctionServer* with identifier 1. The *Clients* node contains two *Process* components with active objects 10 and 11 respectively of class *Bidder*.

4.2 The Class Diagrams

The class diagrams in our model contains active object classes, remote object interfaces and classes that implement remote object interfaces. To distinguish the specific roles of the classes in the applications, we define class stereotype *Active* for active object classes and interface stereotype *IDL* for remote interfaces. An active object is self-executable. To specify the starting point of the execution, we define method stereotype *Main*. An *Active* class must contain one and only one *Main*-stereotyped method, which starts the execution. We also define a similar method stereotype called *Thread*. A *Thread*-stereotyped method is not self-executable. However, when it is called, it will execute on its own execution thread, running concurrently with its caller. A *Thread*-stereotyped method must have neither output nor return parameters.

Figure 2 shows the class diagram for the auction example. Here, an instance of class *Bidder* is an *active* object, and its attribute *serverC* is a *Stub*. In the statechart diagrams, it will be bound to an *Auction* remote object and thus become a client of the object. Similarly, the attribute *interServer* in an *AuctionServer* object will become a client of the other *Auction* remote object for server synchronization.

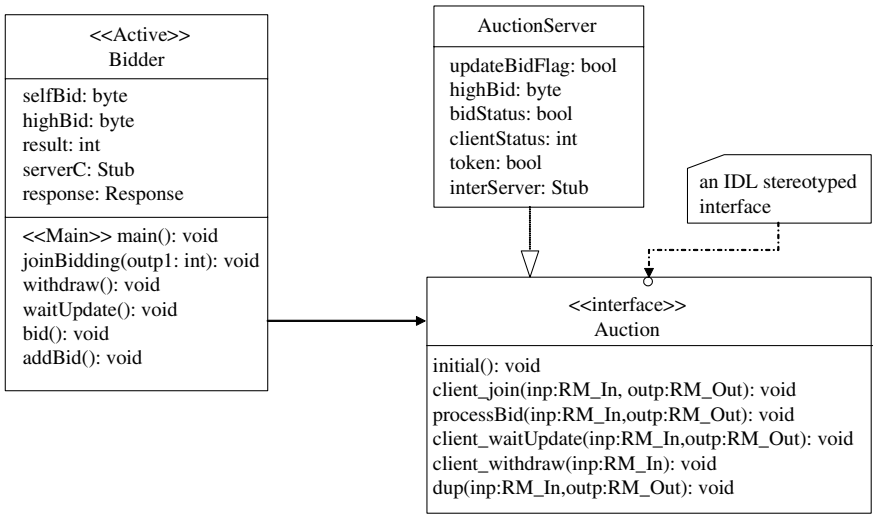

Fig. 2. Class diagram in online auction example

Note that these diagrams are for illustration only. Our CUP tool currently works only on text-based, specially formatted input.

As shown in this figure, to keep the verification model manageable, we uniform the parameters of the remote methods such that: (i) The method must have no return value; (ii) The method must take an *RM_In* object as its input parameter and possibly an *RM_Out* object as its output parameter. As default, the data classes *RM_In* and *RM_Out* contain two integer parameters each. Designers can override these classes if needed, but the parameters must be of type integer or its subtype.

With the restriction, the interface to any remote method is statically determined. That, combined with the statically created remote objects, greatly simplifies the realization of the binding and the remote method invocation process.

As shown in the figure above, two pre-defined CORBA-related classes, namely *Stub* and *Response*, are used to facilitate remote method invocations. A *Stub* object represents a client to a remote object. The *Request* object in CORBA, which is used for dynamic method invocation, is omitted in our model since the restrictions we put on the model reduced much of the dynamic method invocation tasks. For simplicity, the core of the dynamic method invocation, namely the deferred remote method calls, is packed into the *Stub* object. The *Response* class is for getting the result of deferred remote method calls.

Figure 3 shows the details of the class definitions of the *Stub* and *Response*.

The *Stub* class contains six methods: two binding methods, three remote call methods and an *unbind* method.

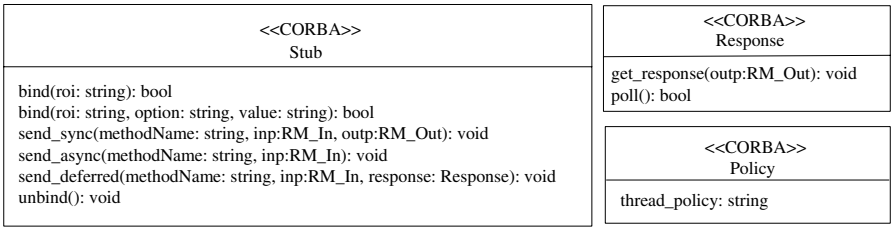

Fig. 3. The Stub and Response classes

A *Stub* object becomes a client to a remote object through its binding methods. The first parameter in both methods defines the type of the remote object to which the *Stub* object will be bound. Its value must be set as one of the *IDL*-stereotyped interface name.

- The binding method with no binding option will randomly bind the *Stub* object to a remote object which implements that interface.
- If the *option* parameter is given value *POA*, the method will bind the *Stub* object to a remote object component implementing the given interface in the specific POA component. The parameter *value* indicates the id of that POA.
- If the *option* parameter is given value *OBJ*, the method will bind the *Stub* object to a specific remote object. The parameter *value* indicates the id of that object. The object must implement the given interface.

After binding, the remote method calls to that object can be made through the *Stub* object. The remote methods can be called synchronously through the *send_sync* method, asynchronously through the *send_async* method or deferred synchronously through the *send_deferred* method. The parameter *methodName* in these methods specifies the remote method that should be called. The parameters *inp* and *outp* contain input and output parameters respectively. The *Response* object is for getting result of deferred method calls. The result of the method invocation can be pulled at any time by calling the *get_response* method of the *Response* object.

A *Stub* object can also be disconnected from the remote object by calling the *unbind* method. After disconnection, the bind methods can be called again to bind the object to a remote object.

Another class shown on Figure 3 is the *Policy* class. Instances of this class define the thread policy for ORB or POA components. (See Figure 1).

A *Policy* object contains a string attribute *thread_policy*, which defines the thread policy of a POA or ORB component. The legal values of *thread_policy* for POA are *thread-per-client*, *thread-per-obj*, *thread-per-POA*, *thread-per-ORB*, *thread-pool(n)* and *thread-per-request*. The legal values for ORB are *single-threaded*

and *multi-threaded*. We assume a multi-threaded ORB has unlimited thread resource and will never block. We also do not consider the inter-ORB communication at this stage.

4.3 The Statechart Diagrams

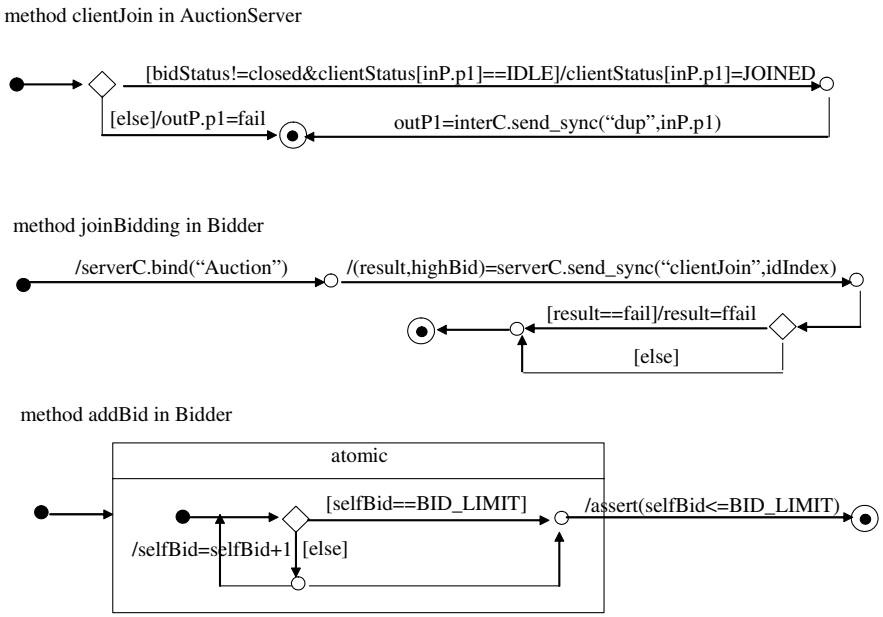

Fig. 4. Partial statechart diagram in online auction example

As we mentioned in the Introduction, in the design model, a statechart diagram defines the behavior of a method. It is built upon parameterized method calls, change events, guard conditions and value assignment actions. Unlike in traditional state diagrams, in UML statechart diagrams, a transition with a guard condition is executable only if the condition is true at the time the system leaves the starting state of the transition. Correspondingly, change event is defined in UML statechart diagrams: A transition with a change event is executable whenever the boolean condition used in the event becomes true. Both cases are reflected in CUP.

In the statechart diagram of method *joinBidding* (the second statechart diagram) in Figure 4, a *serverC* stub is bound to an arbitrary remote object of *AuctionServer*. With this binding, remote method *clientJoin* (see the first statechart diagram) is called in synchronous mode.

We eliminate all concurrency elements in a statechart diagram (sync, fork, join, concurrent regions in composite states). Instead, the concurrency is achieved through calls to *Thread*-stereotyped methods when necessary.

To reduce the verification complexity of the model, we add special semantics for composite states named *atomic*: they identify atomically executed blocks. For example, in the statechart diagram of method *addBid* in Figure 4, since generating a new bid is a sequence of local actions which do not involve any shared data, they are specified as an *atomic* composite state for an atomic sequence of actions. It reduces the verification complexity without any side-effect.

For verification purpose, we add special semantics for states whose name starting with *end*, *progress* or *accept*. They will be translated into their corresponding Promela labels [6], which play important roles during the verification. We also allow users to insert Promela assertion statements in the transition actions as correctness properties. For example, in the statechart diagram *addBid* in Figure 4, the assertion statement verifies if the new bid generated will be no greater than the pre-set price limit.

Due to space limit, we will not discuss statechart diagrams in more details here (cf. [1]).

5 Promela model for CORBA Distribute Object Systems

In the previous section, we have introduced our adaptation of UML for CORBA-based applications. In this section, we explain the corresponding Promela model. Basically, the Promela model is generated according to the following rules:

- Each POA component is translated into a global channel which holds remote method call requests and a set of *servant thread processes* according to its thread policy. The servant threads repeatedly remove the request it should process from the POA channel, invoke proper method inline to service the call and send results (if any) through the return channel provided in the request back to the caller.
- An ORB component with *single-threaded* policy is translated into a global channel which holds remote method call requests and a *dispatcher process* which repeatedly removes element from the channel and dispatches it to proper POA channel.
- An ORB component with *multi-threaded* policy is omitted in the Promela model: The remote method call requests are sent directly to the POA channels.
- An active object is translated into a data attribute, a set of method inlines (each corresponding to a method) and a set of Promela process prototype (each corresponding to a *Main* or *Thread*-stereotyped method). A process of a *Main*-stereotyped method prototype will be activated by the *init* process of Promela.
- A remote object component is translated into an attribute of a customized data type and a set of method inlines. One or more servant thread processes

of its POA will be assigned to service remote method invocations to this object, using the data attributes and the method inlines.
- The binding is extracted from ORBs and is realized by a sole Promela process called *cup_bind*.
- The data class *RM_In* and *RM_Out* are translated into two customized Promela datatypes *CUP_RM_In* and *CUP_RM_Out*.
- A *Response* object is translated into an attribute of data type *CUP_Response*, which contains a channel for retrieving the result of a remote method call.
- A *Stub* object is translated into an attribute of data type *CUP_Stub*, which contains two channels and two integers: Channel *invocation* is for sending remote method calls and channel *response* is for getting the result of a synchronized remote method call; Integer *objID* and *stubID* are for the binding purpose.

Table 5 shows the definitions of *CUP_Response* and *CUP_Stub*.

```
typedef CUP_Stub
{
    int stubID, objID;
    chan invocation;    /* for sending remote method calls */
    /* return channel : outputParameters */
    chan response = [0] of {CUP_RM_Out};
}
```

```
typedef CUP_Response    /* for deferred method calls */
{
    chan responseChan = [1] of {CUP_RM_Out};    /* outputParameters */
}
```

Table 1. CUP data types for the Stub and the Response classses

The binding of a *Stub* object is translated into the proper initialization of the invocation channel and the *objID* attribute. We chose to use a sole Promela process instead of one for each ORB to handle all binding requests and each binding request is serviced by an atomic block. This puts less strain on the state space. Since the variables modified by the binding process is local to the stub or to the generated servant thread, such implementation is free from side-effect.

All binding requests are sent through a global channel *cup_bind_request*. The request contains three *mtype* integers representing the unique id of the *Stub* object, the type of the remote object and the binding option (CUP_RANDOM, CUP_POA or CUP_OBJ). It also contains an integer for POA or remote object id.

The binding process interprets the requests, finds a proper remote object according to the binding options and returns the proper invocation channel together with the remote object id to another global channel *cup_bind_result* to be

picked up by the requester. If the remote object is managed by a POA which belongs to a *single-threaded* ORB, the request channel of that ORB will be sent back. Otherwise, the request channel of the POA will be sent back. If the chosen object is managed by a *thread-per-client* policy POA, the binding process also starts a servant process to service that stub.

The following shows the definition of the binding channels and the translation of the binding for a *Stub* object *stub*:

$chan\ cup\_bind\_request = [0]\ of\ \{mtype, mtype, mtype, int\};$
$chan\ cup\_bind\_result = [0]\ of\ \{chan,\ int\};$

stub.bind("Auction") is translated into

$cup\_bind\_request!stub\_T.stubID, CUP\_IDL\_Auction, CUP\_RANDOM, scratch;$
$cup\_bind\_request?stub\_T.invocation, stub\_T.objID;$

Note that each variable name may be assigned with some suffixes during the translation. As such details are omitted here (cf. [1]), we use T to represent such possible suffix here for readability.

A remote method call is translated into sending of a remote method call element to the invocation channel. For example, suppose an invocation request for method m with input parameter *ip* is made through a *Stub* object named s. If the method has an output parameter *op*, then UML synchronous remote method call

s.send_sync(m, ip, op)

will be translated by CUP into

$s\_T.invocation!s\_T.stubID, CUP\_SYNC, s\_T.response, s\_T.objID,$
$CUP\_METHOD\_m, ip\_T);$
$stub\_T.response?op\_T;$

The element contains the id for the stub object, the synchronization flag, the response channel, the remote object id, the remote method id and the input parameter. Here, CUP_METHOD_m is the generated unique identifier for method m.

Each remote method call element is extracted from the invocation channel by a servant process, interpreted and serviced. The functionality and the quantity of the servant threads are determined by the thread policy of POAs:

- A servant process is created for each ORB which has at least one *thread-per-ORB* policy POA to service all calls made to the request channel of the ORB (to invoke a remote method managed by one of the POAs).
- A set of servant processes is created for each POA that does not have a *thread-per-ORB* policy. They service specific calls made to the request channel of the POA:
  - *thread-per-client*: A servant process is created for each stub object which is a client of a remote object managed by the POA to service the calls from the stub object.

- *thread-per-obj*: A servant process is created for each remote object managed by the POA to service the calls to the object.
- *thread-per-POA*: A servant process is created to service all the calls to the POA.
- *thread-pool(n)*: A set of n servant processes is created to service all the calls to the POA.
- *thread-per-request*: A servant process is created for each invocation request. The process terminates after the call is completed.

To service a call, a servant thread will first determine a proper inline to invoke, according to the method identifier and the remote object identifier sent as part of the request. Then the inline will be invoked with the input parameter from the request, and the output parameter, if any, will be updated in the inline and finally be sent back to the return channel provided in the request.

Due to the limit of space, we will not discuss the generation of the inlines here.

6 Related Work

To boost the acceptance of formal verification techniques, various researchers have been exploiting ways to loosen the restriction that a system must be described formally for verification purpose. An alternative is to derive formal descriptions automatically from existing software artifacts, so that the software developers can get relief from learning a formal specification language and further writing correct specifications in it. Since UML is well accepted by the software industries as graphical design notations, people have been studying the formalization of UML descriptions. In particular, people have discussed various issues regarding the translation from formalized UML statechart diagrams into PROMELA [10, 11, 14, 13].

Our work also involves such a translation. Its details, which is out of the scope of the present paper, is reported in [1]. The most prominent feature in our translation of UML statechart diagrams (as well as class diagrams and deployment diagrams) is that our focus is application-oriented. While other UML $\rightarrow$ Promela translation tools are available, ours specifically deals with the parameterized remote and local method calls and exempt signal events. We allow rich transition syntax while put much restrictions on states, especially those related to the concurrency control. The result is an efficient translation that translates each statechart into a Promela process or a Promela inline. In this regard, our translation shares some commonality with the Java Pathfinder [5]. Our treatment greatly reduced the number of processes in the translated code. We also distinguish between change events and guard conditions, which is not addressed in other translators. Our focus has been put on introducing the realization of the CORBA distributed object system in Promela, which simplifies the verification task for distributed applications built on CORBA-based middlewares.

The verification of distributed computing environment has also gained much of our attention. Various works have been conducted to verify the correctness

of communication protocols on different layers. For example, the specifications of CORBA ORB (Object Request Broker) properties are discussed in [2]. The descriptions of CORBA GIOP (General Inter-ORB Protocol) in PROMELA can be found in [8]. The motivation behind these works is to check the correctness of the protocols themselves. Our work, on the other hand, is targeted at the correctness of the design models of the applications built on top of the CORBA middleware. In doing so, we have not only abstracted and formalized the CORBA-middleware but also embedded it automatically into the design notations during the translation of the adapted UML diagrams.

There are two main middleware families: the DOS middlewares, and the message-oriented middleware. While we have reported our work for the former, one can also find related work for the publish-subscribe middleware, one of the major message-oriented middlewares [3]. In the work of [3], the authors used *reusable, parameterized state machine models* to define the behavior of a *generic* publish-subscribe middleware, and developed translation tool to transform it, together with the descriptions of the components of the applications, into a specific verification model for SMV. The middleware interfaces we used here are an object-based, generic form in UML to describe the *reusable, parameterized state machine*.

A similar work that involves both the formalism of CORBA and its integration into the design specifications of the applications can be found in [9]. There, the authors have presented a technique on modeling the behavior of CORBA ORB in Finite State Processes (FSP) [12] so that the related model checking tool Labeled Transition System Analyzer (LTSA) [12] can apply. To incorporate the modeling of the ORB into the specification of the design models in UML, the authors suggested to use the *stereotypes* on class diagrams to define the threading policy (single-threaded class or multithreaded class), and on method invocations in statechart diagrams to define the synchronization modes of the calls. In reality, however, the thread policy is defined on ORBs and POAs, and instances of the same class (of the application) may be managed by different POAs possibly with different thread policy. Here we have explicitly defined dedicated components for POAs and ORBs, each with an attribute to specify the corresponding thread policy.

There are quite some model-checking tools available off-the-shelf. We chose SPIN because it is a popular, mature and powerful tool. It allows us to benefit from a lot of important features it provides, e.g. on-the-fly verification, support for both rendezvous and buffered message passing, efficient partial order reduction. Our focus is on the transparency and the efficiency of the translated model. For efficiency, our model is basically static and methods are implemented as PROMELA inlines to reduce resource consumption. If desired, CUP can be modified without difficulty to utilize the extensions of SPIN such as dSPIN [4], which may provide some additional features.

7 Conclusion and Final Remark

CUP is for those users who have little knowledge of formal verification techniques but desire to use them. Especially, we consider applying model checking tools to CORBA-based applications.

The major issue of the tool is how to model the applications so that the chosen model checking tool can work effectively. Even with the effectiveness of SPIN, we still need to make our best effort to reduce the complexity of the generated Promela model, because the difficulty comes from two dimensions: one is the complexity of the application logic, and another is from the realization of CORBA-based middleware.

On the aspect of the application logic, we translated most statechart diagrams into Promela inlines, for which we have to generate method invocation graph and create different suffix for labels, method names and variable names accordingly. We also added special semantics to composite states to identify atomically executed blocks. On the aspect of middleware modeling, we first simplified naming, binding, etc. The naming is omitted and binding request are serviced by a single process with each binding request modeled as an atomic block. Each servant thread is modeled as a Promela process. By minimizing the processes used for remote method calls, the complexity is reduced and the resulting model can remain manageable.

We verified the Promela source file generated by current version of CUP for the auction application on a Toshiba Satellite notebook computer with Microsoft Windows XP, Pentium(R) 4 with CPU 3.06GHz speed and 512M memory. The verification to check deadlock on the model with two bidders passed in 160 minutes in exhaustive checking, and less than 7 minutes in supertrace checking with over 99% coverage. We also verified some LTL formulas while polishing the auction application. We have identified errors in verifying:

- One and only one of the bidders should finally succeed and the others should finally fail.
  #define ppp ((Bidder[0].result == fsucc)& (Bidder[1].result == ffail)
  #define qqq ((Bidder[1].result == fsucc)& (Bidder[0].result == ffail)
  LTL: <>(ppp—qqq)
  error: Both bidder fails.
  reason: Bidder may not bid at all before withdrawing.
  solution: Let bidder bid at least once before withdrawing.
- The success bidder should hold the highest bid.
  #define rr1 ((Bidder[0].result == fsucc)&(Bidder[0].selfBid== Auction[0].highBid))
  #define rr2 ((Bidder[1].result == fsucc)&(Bidder[1].selfBid== Auction[0].highBid))
  LTL: <>(rr1|rr2)
  error: the final bid is higher than the *highBid* in *Auction Server*.
  reason: error in auction server synchronization.
  solution: change made in the *dup* method.

Compared with the simplified version of our work where middleware is not modeled and the remote objects are treated as access-free local objects, CUP

has apparently added some overhead to the execution cost to handle each client thread and each remote method call issued by a client thread. This overhead is determined by many factors. Intuitively, the impact of the overhead on handling the client thread decreases with the increase of the workload of the client thread, and the impact of the overhead on handling the remote method call header increases with the frequency of the remote calls: If the clients makes frequent yet short remote method calls, it will cause a great deal of overhead.

We have run the auction example with two bidders under the thread-per-client mode, and with different pre-set price limits 2, 3, 4, 5, 6. See below:

Price Limit= 6 → Depth= 628 States= 5.5e+007 Transitions = 7.23034e+007
Price Limit = 5 → Depth= 586 States= 3.1e+007 Transitions = 4.0888e+007
Price Limit = 4 → Depth= 570 States= 1.4e+007 Transitions = 1.86401e+007
Price Limit = 3 → Depth= 566 States= 5e+006 Transitions = 6.79471e+006
Price Limit = 2 → Depth= 531 States= 1e+006 Transitions= 1.3985e+006

The workload of the client threads and the number of remote method calls in each client thread increase with the pre-set price limit. From the above data, we can see that the impact of the thread overhead (¡e+006) is negligible when the price limit increases. On the other hand, the overhead on the remote method calls remains constant and cannot be ignored.

The state explosion problem due to the increased complexity of the application itself is unavoidable in CUP: typically, the number of states will increase exponentially with the number of active servant threads. For example, if three bidders instead of two compete for an item with price limit 3, the number of state increases rapidly to 3.5e+008.

8 Acknowledgements

The author would like to thank the anonymous reviewers for helpful comments on the preliminary version of this paper submitted to SPIN 2004. This work is supported in part by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN 209774.

References

1. H. Cui. Correctness of distributed systems with middleware. Master's Thesis. School of Computer Science, University of Windsor, April 2003.
2. G. Duval. Specification and verification of an object request broker. In *Proc. of the 20th International Conference on Software Engineering*, pages 43–52, 1998.
3. D. Garlan, S. Khersonsky, and J. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN'03), LNCS 2648*, pages 166–180, 2003.
4. J. Hatcliff, X. Deng, M. Dwyer, G. Jung, and V. P. Ranganath. An integrated development, analysis, and verification environment for component-based systems. In *Proc. of the 25th International Conference on Software Engineering*, pages 160–173. IEEE, 2003.

5. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, April 2000.
6. G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.
7. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
8. M. Kamel and S. Leue. Formalization and validation of the general inter-ORB protocol (GIOP) using Promela and Spin. *Software Tools for Technology Transfer*, 2(4):394–409, 2000.
9. N. Kaveh and W. Emmerich. Deadlock detection in distributed object systems. In *Proc. of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 44–51. ACM Press, 2001.
10. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11:637–664, 1999.
11. J. Lilius and I. Paltor. Formalizing UML state machines for model checking. In *Proc. of the 2nd International Conference on Unified Modeling Language (UML'99), LNCS 1723*, pages 430–445. Springer-Verlag, 1999.
12. J. Magee and J. Kramer. *Concurrency: Models and Programs – From Finite State Models to Java Programs*. John Wiley, 1999.
13. E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing statecharts in PROMELA/SPIN. In *Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 90–101, 1998.
14. T. Schafer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 47:1–13, 2001.