

DEVS-Based Modeling and Simulation of the CORBA Portable Object Adapter

Emmanuelle de Gentili, Fabrice Bernardi, Jean-François Santucci
University of Corsica
UMR CNRS 6134,
Quartier Grossetti,
20250 Corte, France
{gentili, bernardi, santucci}@univ-corse.fr

Keywords:

Modeling, Simulation, DEVS, CORBA, POA

ABSTRACT: *We present in this paper our approach for the modeling and the simulation of distributed architectures based upon Common Object Request Broker Architecture (CORBA). We point out a global methodology for the modeling of algorithmic functions using a DEVS-based approach, and we apply it to a function of the CORBA Portable Object Adapter: find_POA(). We will expose the methodology we defined, which allows the taking into account of all the computer control structures, and then will introduce three types of ports allowing us to model the internal function sequencing. The originality of this paper lies in the fact that we use DEVS theory in order to simulate a Software Object. The modeling is started from the source code of the function and the simulation is validated performing the verification of the followed path.*

1. Introduction

Nowadays, more and more applications are build in order to be executed over a network. These applications are called “distributed applications”, and are based upon a distributed architecture allowing the dialog between objects over such a network.

The CORBA Architecture (Common Object Request Broker Architecture) is one the most important distributed architecture used. This technology provides a flexible, extensible, platform-independent and language-independent framework that is suitable for large-scale deployment. It uses standard CORBA Interface Definition Language based Application Programming Interfaces and CORBA Object Request Broker (ORB) to provide the necessary location transparency and language independence (see [1]). These CORBA components allow running distributed applications on different platforms and written in different languages just by writing Object-Oriented Applications.

In order to study the behaviour of a CORBA-based distributed application, and more precisely of its objects over the network, we initiate an original approach based on the discrete event modeling and simulation. This approach will allow us to simulate the behaviour of objects involved in the CORBA Architecture.

We presents in details in this paper the modeling of one of the functions of the Portable Object Adapter (POA) which is a fundamental of CORBA (You can note that all the other functions have been modelled too, but we can not present them here). The modeling of the behaviour has been performed using the DEVS

Formalism (Discrete Event Specification) starting from the source code written in C++.

The first part of the paper presents the fundamentals of the DEVS modeling and simulation.

The second part presents the global methodology we developed for the modeling of algorithmic functions. This methodology is illustrated with a little example.

Then, we will present very briefly the CORBA Portable Object Adapter and the modeling and the simulation of one of its functions: the find_POA() function.

The two last parts will be devoted to the presentation of the implementation of the simulation and to a conclusion and some perspectives of work.

2. Behavioral DEVS Modeling

The basic modeling approach is based upon the multi-views notion. This concept allows a gradual introduction of the complexity of a system by its analysis according to various points of view. Each model obtained represents a particular portion of the specifications of the global model. The two main kept views are the behavioural view and the structural view. In this paper, we will use only the behavioural view.

The behavioural elements, as the structural components, are inter-connected so as to be able to exchange information through inputs-outputs sets. For our modeling approach using DEVS formalism (Discrete Event System specification), they are of two types:

Atomic Models: They are the basic components of the behavioural view. They provide a local description of the dynamic behaviour of a system. They are described by the following elements:

AM = $\langle X, Y, S, t_a, \delta_{int}, \delta_{ext}, \lambda \rangle$ with:

- X: set of inputs
- Y: set of outputs
- S: set of state variables
- t_a : time advance function
- δ_{int} : internal transition function
- δ_{ext} : external transition function
- λ : output function

Coupled Model: This kind of components corresponds to a set of behavioural elements (Atomic Models and/or other Coupled Models). They allow the description of the manner a new component is created by interconnecting some others. They are described by the following elements:

CM = $\langle X, Y, C, LCS, EIC, EOC, IC, L \rangle$ with:

- X: set of inputs
- Y: set of outputs
- C: list of the constituting components
- LCS: link to the associated structural component
- EIC: external output coupling
- IC: internal coupling
- L: priority list on the sub-components

Starting from these definitions, we can perform a simulation of the modeled discrete-event elements using a simulation tree created automatically. Full concepts are provided in [2], [3], [10], [11] and [12].

The next part of the paper presents how these concepts are used in order to find a global methodology for the modeling of algorithmic computer functions.

3. Global Methodology for Algorithmic Functions Modeling

3.1. Overview of the Methodology

Algorithmic functions can be seen as discrete-event

systems, and then can be theoretically modeled and simulated using DEVS approach. However, we found that a generic methodology can be applied, starting from the source code of the function. The basic approach is to consider algorithmic functions as coupled models, only composed by atomic models sequentially connected. We consider that an atomic model represents the modeling of the behaviour of the function between two calls to other external functions.

Internal variables of the function are binded to state variables inside an atomic model, and these variables are carried through specialized ports as long as the functions, and therefore the other atomic models, need them. This is viable since we start the modeling from the source code of the function.

For a better understanding, we introduce three types of ports allowing the connection between two or more models (Figure 3.1):

- *Sequencement Ports:* these ports are atomic models ones, and are used to carry the state variables between two models. They can connect only atomic models inside a coupled model.
- *Call Ports:* these ports are used by atomic and coupled models. They carry the name of the function to be called. They connect two coupled models, therefore two functions.
- *Parameter Ports:* these ports are used to carry the parameters needed by a function. They are similar to Call Ports, since they connect two functions and are used by the two kinds of models.

Input ports of the coupled models are constituted by the ports binded to the function parameters (Parameter Ports) and by a Call Port used to ensure that the function is really the one to be activated. Output ports are composed by a port binded to the return variable of the function and another dealing with a possible computer exception. Atomic models are more complicated since they must deal with the parameters of the function, but also with the sequencement of the state variables.

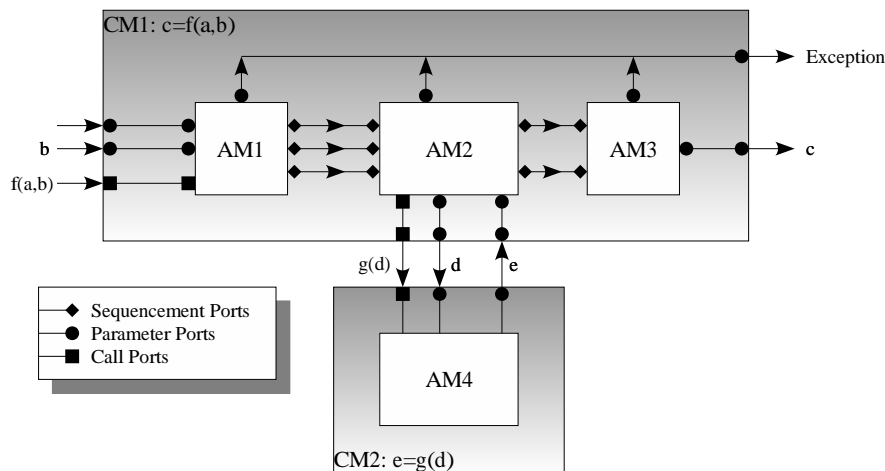


Figure 3.1: Illustration of the global methodology

3.2. Control Structure Modeling

Most algorithmic functions use control structures such as “if...then...else” or the “for” statement.

- the “if” statement (Figure 3.2): AM1 contains the test result as a state variable. The orientation will be performed using the output function of the Atomic Model. The “case” statement is similar.

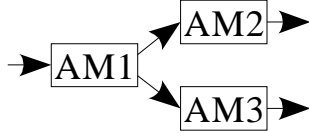


Figure 3.2: Illustration of the “if” Statement Modeling

- The “for” statement (Figure 3.3): AM1 contains the test result and the increment value as state variables.

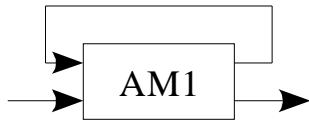


Figure 3.3: Illustration of the “for” Statement Modeling

- The “while” statement (Figure 3.4): AM1 contains the test value as state variable. The update is performed using the AM2 lambda function.

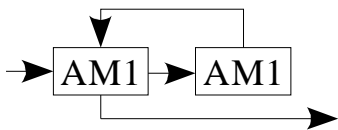


Figure 3.4: Illustration of the “while” Statement Modeling

- The “do” statement (Figure 3.5): The code between “do” and “while” is executed at least at one time. Then, the test result is placed as a state variable of AM2.

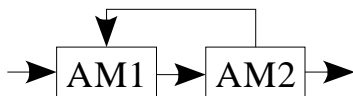


Figure 3.5: Illustration of the “do” Statement Modeling

3.3. Simple Example

Figure 3.6 presents a very simple function and its modeling using our approach. The function is associated with a Coupled Model presenting two input ports (a parameter port and a call port) and an output parameter port. It can be modeled using six very simple Atomic Models.

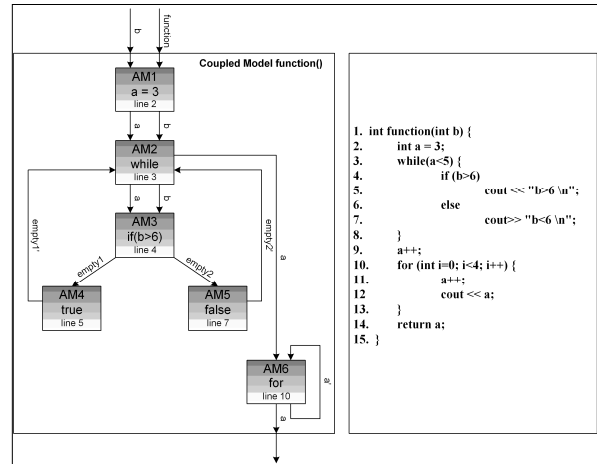


Figure 3.6: Modeling of a Simple Function

AM1 declares an “a” state variable, initialises it and send it through a sequencement port. Following models will need the “b” variable, so it is sent too.

AM2 contains the result of the test in line 3 and will orientate the variables towards AM3 or AM6.

AM3 contains the result of the test in line 4 and will send an empty message towards AM4 or AM5 in order to continue the simulation, even if there are no variables to be passed.

4. CORBA Portable Object Adapter (POA) Modeling

4.1. The CORBA Portable Object Adapter

Figure 4.1 shows the significance of the Portable Object Adapter (POA) in a CORBA-based distributed architecture (see [4] and [5]).

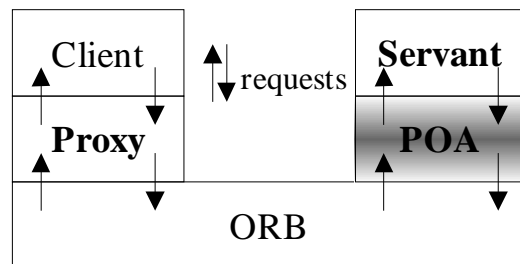


Figure 4.1: Position of the POA in the CORBA Architecture

The CORBA architecture is based upon the Object Request Broker (ORB), which carry the requests from the client object to the server one. The interface between the client and the ORB is complex and we will note it “Proxy”. The POA is the interface between the server objects and the ORB.

A POA is connected to many servants, which are object implementing interfaces of operations defined using the Interface Definition Language (IDL). In object-oriented

languages (C++, java), servants are implemented using one or more objects. A client never interacts directly with a servant, but always through an object. Each servant is referenced thanks to an Object Identifier (ObjectId), identifying an object within the scope of its own Object Adapter. The mapping is made using an Active Object Map (AOM) maintained by each POA and providing the way to access the currently active objects using servants. Functionnalities of the POA can be described as follows (see [8] and [9]):

Generating Object References: the POA is responsible for generating object references for the CORBA objects registered with it. Object references identify a CORBA object and contains addressing informations that allow clients to invoke operations on that object in a distributed system.

Activation and deactivation of servants: the POA can activate CORBA objects to handle client requests. Similarly, it can deactivate objects and destroy their corresponding servants when they are no longer needed.

Demultiplexing requests to servants: when the ORB receive a request from a client, it collaborates with the POA to ensure that the request reaches the proper servant. The POA parses the request to locate the ObjectId of the servant, which it uses to locate the correct servant and invoke the appropriate operation.

Invoking servant operations: the operation name is specified in the CORBA request. Once the POA locates the target servant, it dispatches the requested operation on it.

One of the main advantages of the POA is that it allows programmers to construct servant that are portable between different ORB implementations. Another is that it allows servants to assume complete responsibility for an object's behavior.

A server application may create multiple POAs to support different kinds of CORBA objects. In this case, a POA Manager encapsulates the processing state of one or more of these POAs.

When a request is received for a child POA that does not yet exists, the ORB will invoke an operation on an \Adapter Activator which will decide whether or not to create the required POA on demand.

In the next section, we present the modeling of one the numerous functions composing the OMG Portable Object Adapter: the find_POA() function.

4.2. Modeling of the find_POA() Function

Our main hypothesis for modeling is to consider that the POA is created and correctly supplied by the ORB. Global variables of the implementation can then be considered as state variables of the ORB coupled model.

We choose to present in this paper the find_POA() function. This function returns a pointer to a POA adapter name if it is a child of the target POA. If the target POA has no child of the specified name and activate is set to TRUE, find_POA() invokes the target POA's adapter activator, if one exists. The adapter activator attempts to restore POA adapter; if successful, find_POA() returns the specified POA object. If no POA is returned, the function raises the exception AdapterNonExistent.

The algorithmic/C-based form of find_POA() can be written as following (see [6] and [7]):

```

1. POA_ptr find_poa(adapter, activate)
2. {
3.   if (getDestroyed())
4.     throw Exception;
5.   bool check=true;
6.   if (containsKey(adapter)&& activate)
7.   {
8.     adapterActivator =
9.       getAdapterActivator();
10.    if (adapterActivator != NULL)
11.      check =
12.        unknownAdapter(adapter);
13.   }
14.   POA poa;
15.   if (check)
16.     get(adapter, poa);
17.   if (poa == NULL)
18.     throw AdapterNonExistent;
19.   return poa;
20. }

```

It is important to point out that this representation is a simpliflicated one and is presented like this only for comprehension.

This function accepts two parameters (adapter and activate) and sends back a POA_ptr. Starting from this, we can build the Coupled Model ports: X=<adapter, activate, find_poa(>. "adapter", "activate" are parameter ports while "find_poa()" is a call port. To these ports, we add two new ports (poacontrol and children) that are used as global variables and which we consider as state variables of the ORB. Finally, the input ports of the Coupled Model find_POA() appear to be the following: X=<adapter, activate, children, poacontrol, find_poa(>.

The output ports are very easy to define; we create a port for the return value and a port for a possible exception during the execution: Y=<Exception, POA_ptr>. The Coupled Model input ports are binded with the ports of the first met Atomic Model.

Line 3 defines an "if" statement using the result of a getDestroyed() function as test. So, we build an atomic model (AM1) containing the test result as a state variable. The input ports are binded to the Coupled Model ones and the output ports are sequencements ports that will carry the state variables along the following Atomic Models. In order to evaluate the test, AM1 is binded to another Coupled Model called "getDestroyed"

which will return a Boolean value. We add then two new ports to the find_POA() Coupled Model, one output call port towards getDestroyed() and one input parameter port for the returned value.

Line 5 defines a Boolean “check”. This is done in our modeling approach using an Atomic Model (AM2) that will have to pass this variable through its ports.

AM3 is the Atomic Model corresponding to line 6 (“if” statement). It is linked to two other Atomic Models using

sequencement ports (AM4 and AM6) and to another Coupled Model containsKey(), using two parameter ports and a call one.

Figure 4.2 gives the whole Coupled Model. It shows all the Atomic Models defined and the links to the external needed Coupled Models. The input and output ports of the main coupled model are supposed to be coupled with the whole ORB model.

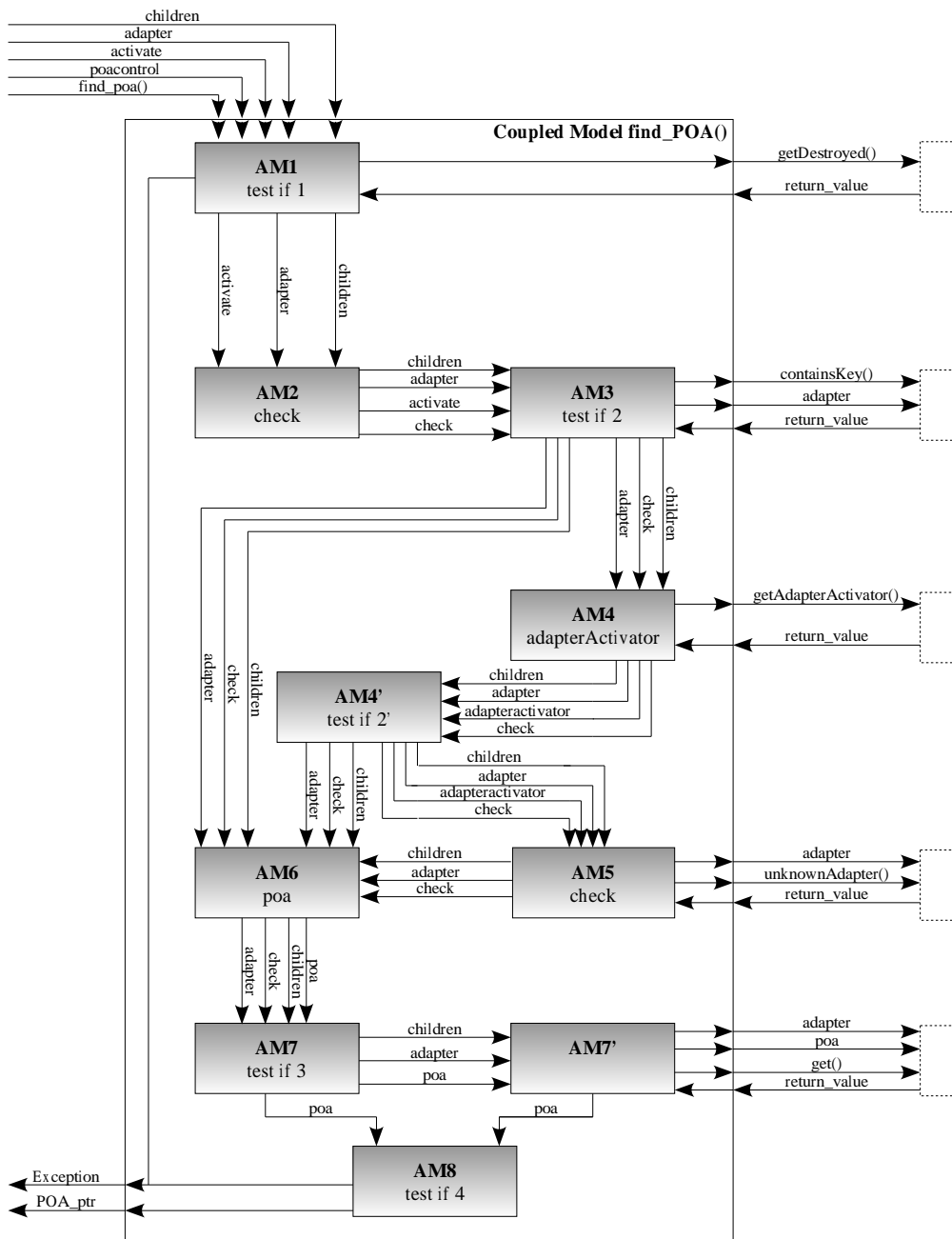


Figure 4.2: Modeling Scheme of the find_POA() Function

5. Implementation of the Simulation

Implementation of the simulation based upon the modeling scheme presented before has been performed using a Java-based simulation engine.

The simulation allowed us to compare the behaviour of the model to the behaviour of the simulated function.

The validation is achieved by a complete path comparison, and results appeared to be very acceptable.

We presented in this paper only the `find_POA()` function, but we performed the simulation for all the Portable Object Adapter methods, and all appeared very conclusive.

6. Conclusions and Perspectives of Work

This paper deals with one the first parts of our CORBA Portable Object Adapter modeling and simulation. We choose to present here the `find_poa()` function which is one the methods of the POA object. This function is a good example since it allows to show a great part of the generic methodology we defined.

The DEVS-based approach for modeling and simulation is usually applied to physical systems. We built a methodology allowing to use this approach for computer functions. This is very interesting for the simulation of distributed applications based upon the CORBA architecture which will be more and more used. Our first step is to model the whole CORBA architecture, in order to be able to model then the higher level applications and simulate them. Then, we will be able to add a physical network simulation tool in order to simulate the distributed execution of an application over a network.

The modeling and the simulation are achieved starting from the source code of the functions to be simulated, using the methodology described previously. The main problem we meet is for the validation.

Two ways are exploitable: the first one is to use a debugger software to follow the function running step by step and comparing the results to those obtained from the simulation. The second one is to determine all possible paths and to follow them. When all paths give good results, we can consider that the model of the function is validated.

The main inconvenient of our approach can appear to be the building of big coupled models for a simple function. This is not a real problem in effect, since we use some

graphical tools providing us a fast implementation of the whole models. Another point is that the atomic models are numerous, but also usually quite simple.

We have a lot of perspectives of work. The first one is to achieve and validate the complete Portable Object Adapter model and the ORB one. Starting from these two basic elements of the CORBA architecture, we will be able to model and simulate distributed applications over a network using a commercial physical network simulation tool. However, the main objective of our work is based upon a collaboration with ALCATEL-Marcoussis, and will consist in the modeling and the simulation of new CORBA services.

7. References

- [1] D. Acreman, G. Moujeard, and L. Rousset, "Développer avec CORBA en Java et C++", Campus Press référence, 1999.
- [2] A. Aiello, "Environnement Orienté Objet de Modélisation et de Simulation à Evènements Discrets de Systèmes Complexes", PhD Thesis, University of Corsica, 1997.
- [3] M. Delhom, "Modélisation et Simulation Orientées Objet, Contribution à l'Etude du Comportement Hydrologique d'un Bassin Versant", PhD Thesis, University of Corsica, 1997.
- [4] J.M. Geib, "CORBA: Des concepts à la pratique", InterEditions, 1998.
- [5] OMG, "Corba 2.4, Specifications of the Portable Object Adapter", 2000.
- [6] OOC, Inc., "Orbacus for C++ and Java", 1999.
- [7] OOC, Inc., "CORBA/C++ Programming with Orbacus, Student Workbook", 2000.
- [8] D.C. Schmidt and I. Pyarali, "Object Interconnexions", In SIGS C++ Report Magazine, Columns 11 to 14, 1997
- [9] D.C. Schmidt and I. Pyarali, "An Overview of the CORBA Portable Adapter", In ACM Standard View on CORBA, 1998.
- [10] B.P. Zeigler, "Theory of Modeling and Simulation", Academic Press, 1976.
- [11] B.P. Zeigler, "Object-Oriented Simulation with Hierarchical, Modular Models", Academic Press, 1997.
- [12] B.P. Zeigler, "Theory of Modeling and Simulation, 2nd Edition", Academic Press, 2000.