

The DSSV Methodology: High Level Validation of CORBA Architecture using a Discrete Event Simulation Approach.

Emmanuelle de Gentili
Fabrice Bernardi
Jean-François Santucci

University of Corsica
UMR CNRS 6134
Quartier Grossetti, BP.52
20250 Corte, France

ABSTRACT

Several commercial software allow to model and simulate an architecture at the physical network layer. However, none of them allow simulating the operating scheme of a distributed object architecture. The aim of this article is to propose a methodology based on the DEVS formalism for the modeling and the simulation of distributed object architectures. We extend the classical “Waterfall” software model with three different description layers: the informal specification layer, the formal specification layer and the code layer. The first layer describes a software using a textual, human-readable approach. The second one describes a software using an approach combining, for instance, an UML description and little pieces of code for some precise details of the implementation. Finally, the third layer describes a software only using code. We will see that this methodology occurs at the formal description layer.

1 INTRODUCTION

Distributed object architectures are generally large and complex pieces of software. Predicting the behaviour of such a system can appear to be a very hard task, especially in large-scale applications. Several commercial tools allow the modeling and the simulation of a network, but only in the lowest layers of the classical OSI model. However, none of them propose to simulate a distributed object architecture before the real implementation. We think that modeling and simulating such an application can bring many benefits in terms of software conception costs, financial cost and reliability of the concrete application. Starting from these considerations, we propose in this article a methodology called DSSV (Discrete event System Specification and Validation) based on the DEVS formalism (see Aiello (1997); Zeigler et al. (2000)) and allowing the modeling and the simulation of distributed object architectures before the real implementation.

Our basic approach is to extend the “Waterfall” software model (see Sommerville (2001)) with three different description layers for software: the informal specifica-

tion layer, the formal specification layer and the code layer. The first layer describes a software using a textual, human-readable approach. The second one describes a software using an approach combining, for instance an UML description (Muller and Gaertner (2000); Booch et al. (1998)) and little pieces of code for some precise details of the implementation. Finally, the third layer describes a software only using code. We will see that this methodology occurs at the formal description layer.

The paper is organized as follows: in Section 2, we introduce the precise aim of this article. We present the modeling, the simulation and the validation of complex systems and the three description layers of a software. We introduce also our methodology for the modeling and the simulation of a distributed object architecture. Section 3 introduces in this part the DEVS formalism. The first part presents the modeling process and the two basic elements of this approach. The second part presents the simulation process. We will see that one of the most important points is that the simulator is built directly from the model. Section 4 is devoted to the presentation of DSSV. We will introduce the basics of the methodology, the rules and section. Section 5 presents an example of DSSV modeling. This example concerns the find-POA() function of the CORBA Portable Object Adapter. We will introduce the central position of the POA in the CORBA architecture, the formal and informal specifications of the find_POA() function and the modeling and the simulation using the DSSV approach. Finally, Section 6 concludes this article and provides some perspectives of work.

2 PROBLEM STATEMENT

We introduce in this section the precise aim of this article. We present the modeling, the simulation and the validation of complex systems and the three description layers of a software. In the last part, we introduce our methodology for the modeling and the simulation of a distributed object architecture.

2.1 Modeling, Simulation and Validation of Complex Systems

For P.A. Muller, a model is “an abstract description of a system or a process, a simplified representation allowing one to understand and simulate” (Muller and Gaertner (2000)). The process combining identification of a system and its model design is called modeling. This model is combined with a control structure called simulator, allowing to produce a possible system behavior under various conditions. This coupling is called simulation. J. Popper defines simulation as “the activities set that consist on building, testing, validating and analyzing a formal model, elaborated in order to represent the significant aspects of a system” (Popper (1973)).

The needs in modeling and simulation techniques come from three fundamental motivations:

- behavioral prediction: the goal is to anticipate how a system reacts when facing external stimuli;
- control: the goal is to manage a system by acting on one or more of its sub-systems;
- existing systems improvements or new systems building.

Therefore, The modeling and simulation process induces three main entities: the system, the model and the simulator (Zeigler (1976)).

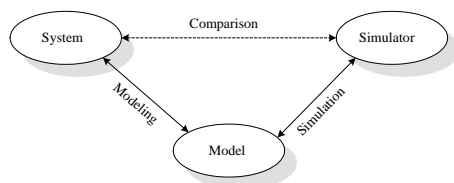


Figure 1: The Three Entities of the Modeling and Simulation Process

These three entities are bound by two links (Figure 1). The first one is the modeling link that bound the system and its model. It represents the data communication between these entities. The second one is the simulation link that bound the model and the simulator. It represents the data exchange allowing the results generation.

The process of comparing experiment measurements with simulation results is called validation. For H. Vangheluwe, this process must be performed within the context of a certain experimental frame (?). He notes that a large number of matching measurements and simulation results, though increasing confidence, does not prove validity of the model. Thus, the correspondence in generated behavior between a system and its model will only hold within the limited context of the experimental frame.

2.2 The Three Description Layers of a Software

Royce introduced the “waterfall model” in 1970 (?). This model of the software development process map onto fundamental development activities (Figure 2):

1. Requirements Analysis and Definition: the system’s services, constraints and goals are established by consultation with system users.
2. System and Software Design: establishes an overall system architecture.
3. Implementation and Unit Testing: the software design is realized as a set of programs or program units.
4. Integration and System Testing: the individual program units or programs are integrated and tested as a complete system to insure that the the software requirements have been met.
5. Operation and Maintenance: the system is installed and put into practical use.

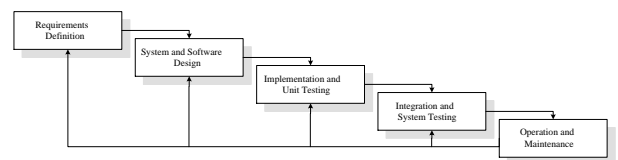


Figure 2: The “Waterfall” Software Life Cycle

We propose to complete this process by introducing three different description layers: the informal specification layer, the formal specification layer and the code layer. The first layer is the informal specification one. It describes a software using a textual, human-readable approach. This layer is used in the first step of the process. The second one is the formal specification layer. It describes a software using an approach combining, for instance UML description (Booch et al. (1998)) and little pieces of code for some precise details of the implementation. This layer is used in the second and third steps of the process. Finally, the last layer is the code one. It describes a software only using code. It is used in the third and fourth steps of the process.

2.3 Distributed Object Architectures High Level Simulation

For I. Sommerville, “a distributed system is a system where the information processing is distributed over several computers rather than confined to a single machine” (Sommerville (2001)). He identifies three main forms of distributed systems: multiprocessor architectures, client-server and P2P (Point-To-Point) architecture and distributed object architectures. We focus in this article on this last point.

In a distributed object architecture, the fundamental system components are objects that provide an interface to a set of services that they provide. Other objects call on these services with no logical distinction between a client and a server. Such an architecture allows the communication between objects distributed over a network using a middleware (Figure 3).

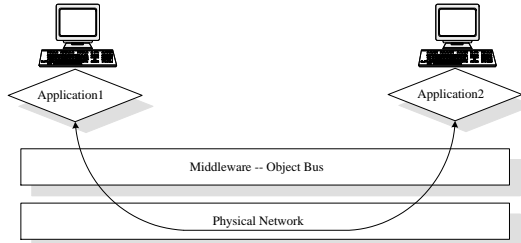


Figure 3: A Distributed Object Architecture

The middleware is generally a large and complex object-oriented framework. This complexity is increased when the user's application is added. We think that modeling and simulating such a complex architecture before implementation can bring many benefits in terms of development costs and in terms of reliability of the whole software that have to be developed.

Several commercial software allow to model and simulate such an architecture at the lowest layer, e.g. the physical network layer. However, none of them allow to simulate the operating scheme of a distributed object architecture. The aim of this article is to propose a methodology based on the DEVS formalism for the modeling and the simulation of distributed object architectures described at the formal specification layer as defined before. This methodology can be used in order to validate the approach before the real implementation.

The basic idea is to consider a formal description of a software as a discrete-event system. Starting from this, we can apply a generic methodology to model and simulate this description using the DEVS formalism.

3 THE DEVS FORMALISM

We introduce in this section the DEVS formalism. The first part presents the modeling process and the two basic elements of this approach. The second part presents the simulation process. We will see that one of the most important points is that the simulator is built directly from the model.

3.1 The Modeling Process

The DEVS (Discrete Event System Specification) formalism introduced by B.P. Zeigler in the early 70's is a set-theoretic formalism which provides a means of modeling

discrete event systems in a hierarchical and modular way (Zeigler et al. (2000)). With this formalism, we can perform modeling more easily by decomposing a large system into smaller component models with coupling specification between them. DEVS defines two kinds of models: atomic models and coupled models.

An *atomic model* is a basic model with specifications for the dynamics of the model. It describes the behavior of a component, which is indivisible, in a timed state transition level. This kind of model has an internal structure which dictates how inputs and states are transformed to outputs or other states. Formally, an atomic model is specified by a 7-tuple:

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

where:

X is the input ports set, through which external events are received.

Y is the output ports set, through which external events are sent.

S is the states set; Two state variables are usually present, "phase" and "sigma". In the absence of external events, the system stays in the current "phase" for the time given by "sigma".

$\delta_{int} : S \rightarrow S$ is the internal transition function; This function specifies to which state the system will transit after the time given by the time advance function has elapsed.

$\delta_{ext} : Q \times X \rightarrow S$ is the external transition function where $Q = \{(s, e) | s \in S, 0 \leq e \leq t_a(s)\}$ is the total state set, and e the elapsed time since the last transition; This function specifies how the system changes state when an input is received. The effect is to place the system in a new "phase" and "sigma" thus scheduling it for a next internal transition. The next state is computed on the basis of the present state, the input port and value of the external event, and the time that has elapsed in the current state.

$\lambda : S \rightarrow Y$ is the output function; This function generates an external output just before an internal transition takes place.

$t_a : S \rightarrow \mathbb{R}_{0,\infty}^+$ is the time advance function, where $\mathbb{R}_{0,\infty}^+$ is the set of the positive reals between 0 and ∞ ; This function controls the timing of internal transitions. When the "sigma" state variable is present, this function just returns its value.

The four elements in the 7-tuple, namely δ_{int} , δ_{ext} , λ and t_a , are called the DEVS characteristic functions.

The second kind of models are the *coupled models*. They tell how to couple several component models together to form a new model. This kind of model can be employed as a component in a larger coupled model, thus giving rise to the construction of complex models in a hierarchical fashion. Formally, a coupled model is specified by another 7-tuple:

$$CM = \langle X, Y, M, EIC, EOC, IC, SELECT \rangle$$

where:

X is the input ports set, through which external events are received.

Y is the output ports set, through which external events are sent.

M is the set of all component models.

$EIC \subseteq X \times \cup_i X_i$ is the external input coupling relation which connects the input ports of the coupled model to one or more of the input ports of the components. This directs inputs received by the coupled model to designated component models.

$EOC \subseteq \cup_i Y_i \times Y$ is the external output coupling relation which connects output ports of components to output ports of the coupled model. Thus, when an output is generated by a component, it may be sent to a designated output port of the coupled model and thus be transmitted externally.

$IC \subseteq \cup_i X_i \times \cup_i Y_i$ is the internal coupling relation which connects output ports of components to input ports of other components. When an input is generated by a component, it may be sent to the input ports of designated components (in addition being sent to an output port of the coupled model).

$SELECT : 2^M - \phi \rightarrow M$ is a function which chooses one model when more than 2 models are scheduled simultaneously.

3.2 The Simulation Process

The components used to describe a specific model are used to automatically generate the corresponding simulator. Three kinds of simulation elements have been defined: the main coordinators, the coordinators and the simulators. These elements are called processors.

Simulators and coordinators are respectively in charge of managing atomic and coupled models. Each atomic model is bound to its simulator and each coupled model is bound to its coordinator. The main coordinator manages the whole simulation process and is bound to the coordinator of the larger coupled model representing the whole system.

The simulation is performed using different kinds of messages that are exchanged between the processors. Four kinds of messages have been defined (Figure 4):

- the “*” messages associated with the internal transitions;
- the “x” messages associated with the external transitions;
- the “y” messages associated with the output functions;
- the “d” messages associated with the simulation process synchronization.

Each one of these messages contain information describing the events to be treated during the simulation. This information is divided in five fields: a type, a “must-arrive” time, a source process, a destination port and a value.

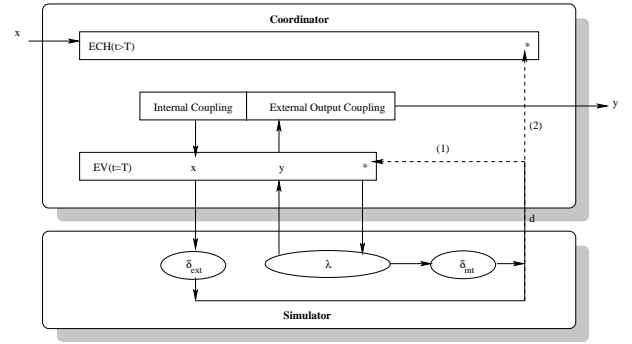


Figure 4: Messages Exchange in the DEVS Simulation Process

A coordinator has two schedulers: the first for the messages which have a carried time t greater than the simulation time T (ECH), and the second one in order to store the messages with a carried time equal to the simulation time T (EV).

When a simulator receives an x message, its external transition function is executed and a d message is sent. This d message is transformed in a $*$ message that leads to the output function which returns an y message. The internal transition function is then automatically executed and can return a new d message.

4 THE DSSV METHODOLOGY

This section is devoted to the presentation of DSSV. We will introduce the basics of the methodology, the rules and some definitions. A first example of modeling is given at the end of this part.

4.1 Basics of the methodology

Algorithmic functions can be seen as discrete event systems, and thus can be theoretically modeled and simulated using the DEVS formalism. However, we found that a generic methodology can be applied starting from the source code of the function combined with, for instance, an UML description. Thus, this methodology occurs at the formal specification layer of a software.

The basic approach consists in considering a software program as a collection of algorithmic functions. These functions are bound to coupled models, only composed by atomic models sequentially connected. We call the DSSV-Methodology (DEVS-based Software Simulation and Validation methodology) a methodology allowing to model and simulate a formally specified software using the DEVS formalism.

In order to define our methodology, we stated some rules and defined different elements necessary for the comprehension.

4.2 The DSSV Rules

We defined four rules to be applied in the methodology:

1. At the same time, only one event can happen on the same port: On the one hand, if the port is an input one, the port is duplicated. On the other hand, only one output is allowed because of the algorithmic sequencing.
2. We consider that an atomic model represents the basic actions of a function between the call of other functions;
3. If a function needs to call an external function, we define an atomic model dedicated to this call;
4. One function is composed by “n” atomic models:

The models set is defined by:

$$M = \{AM_i\}_{i \in \mathbb{N}^*} \cup \{CM_j\}_{j \in \mathbb{N}}$$

The algorithmic functions set is a subset of M :

$$F = \{f_i \subset \{CM_i\}_{i \in \mathbb{N}} / f_i = \{AM_i\}_{i \in \mathbb{N}^*}\}$$

4.3 Special Ports Definitions

The DSSV methodology defines several kind of ports. The first of them are the “sequencing ports” kind allowing establishing some internal interconnections between two atomic models in the same coupled model representing an algorithmic function. These ports are used to carry the state variables between two models. They can connect only atomic models inside a coupled model.

The methodology defines four kinds of ports in order to fully determine the external interconnections between two atomic models belonging to two different coupled models (Figure 5):

- The “call ports” allow identifying the considered function by providing a unique function identifier. These ports can be used by both atomic and coupled models and connect two coupled models, therefore two functions.
- The “parameter ports” are used to carry the parameters of the considered function. They are similar to call ports since they connect two functions and are used by the two kinds of models.
- The “secondary call ports” are call ports dedicated to external functions couplings. They allow “jumping” to another function and are output ports for the principal coupled model.
- The “secondary parameter ports” are parameter ports dedicated to external functions couplings. They carry data between the various coupled models.

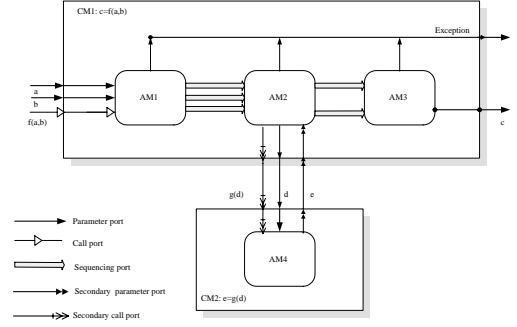


Figure 5: The Special Ports

4.4 Formal Port Sets Definitions

We define:

- P_C as the output call ports set of a coupled model;
- P_P^i as the input parameter ports set of a coupled model;
- P_P^o as the output parameter ports set of a coupled model;
- P'_C as the secondary output call ports set of a coupled model;
- P'^i_P as the secondary input parameter ports set of a coupled model;
- P'^o_P as the secondary output parameter ports set of a coupled model;

The set of the input ports of a coupled model is:

$$P_{CM}^i = P_C \cup P_P^i \cup P'^i_P$$

The set of the output ports of a coupled model is:

$$P_{CM}^o = P_P^o \cup P'_C \cup P'^o_P$$

The full ports set P_{CM} of a coupled model is then:

$$P_{CM} = P_C \cup P_P^i \cup P_P^o \cup P'_C \cup P'^i_P \cup P'^o_P$$

Atomic models introduce two new sets: the input sequencing ports sets P_S^i , and the output sequencing ports sets P_S^o . Then, the full ports set of a DSSV model, noted Π , can be written as follows:

$$\Pi = P_{CM} \cup P_S^i \cup P_S^o$$

4.5 Focus on “Jumps”

We saw that a call to another coupled model is performed when a function needs the results of another one. This call is what we defined as a “jump”. During this jump, the main function remains idled since it must wait for the result of

the other one. This is also implied by the sequencing of the function and of the model.

A “call coupling” is the aggregation of the three coupling existing kinds:

- External input couplings: from the coupled model towards the atomic model.
- Internal coupling: from a coupled model towards another coupled model. The “internal” word refers to the highest-level coupled model representing the whole software.
- External output coupling: from an atomic model towards its parent coupled model.

We see that this kind of coupling can be seen as the formal representation of a jump.

4.6 Modeling Example

Figure 6 presents the DSSV model of the following very simple function.

```

1. int function(int b) {
2.   int a = 3;
3.   while (a < 5) {
4.     if (b > 6)
5.       cout << "b > 6";
6.     else
7.       cout << "b < 6";
8.     a++;
9.   }
10.  for (int i = 0; i < 4; i++) {
11.    a++;
12.    cout << a;
13.  }
14.  return a;
15.}

```

This function is associated with a coupled model presenting two input ports (a parameter port and a call port) and an output parameter port. It has been modeled using six very simple atomic models.

AM1 declares an “a” state variable, initializes it and send it through a sequencing 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 direct 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 resume the simulation, even if there is no variables to be thrown.

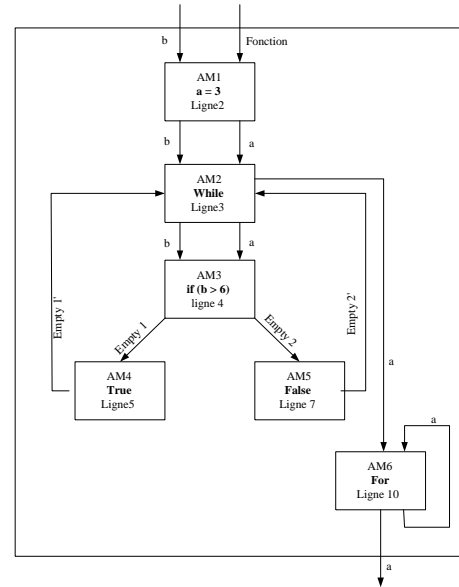


Figure 6: Algorithmic Function Modeling Example

5 A CASE-STUDY: THE find_POA() FUNCTION OF THE CORBA POA

We present in this part an example of DSSV modeling. This example concerns the find_POA() function of the CORBA Portable Object Adapter omg (2001a,b); Pope (1997). We will introduce the central position of the POA in the CORBA architecture, the formal and informal specifications of the find_POA() function and the modeling and the simulation using the DSSV approach.

5.1 The CORBA Portable Object Adapter

Figure 7 shows the significance of the Portable Object Adapter POA) in a CORBA-based distributed architecture.

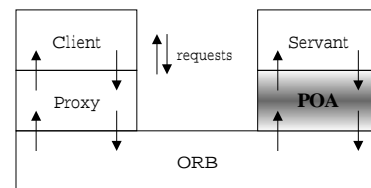


Figure 7: The CORBA Architecture

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 (Geib et al. (1999); Pyarali and Schmidt (1998)).

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 (Schmidt and Vinoski (1997, Schmidt and Vinoski)).

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 the complete responsibility for an object's behavior.

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

5.2 Informal and Formal Specifications of the *find_POA*() Function

The CORBA Portable Object Adapter is implemented as a class presenting some operations like *create_POA()*, *find_POA()*, *destroy()* or *get_servant()*. 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:

```

1. POA_ptr find_POA(adapter, activate) {
2.   if (getDestroyed()) throw Exception;
3.   boolcheck = true;
4.   if (containsKey(adapter) && activate) {
5.     adapterActivator =
6.       getAdapterActivator();
7.     if (adapterActivator != NULL)
8.       check = unknownAdapter(adapter);
9.   }
10.  POA poa;
11.  if (check) get(adapter, poa),
    if (poa == NULL)
        throw AdapterNonExistent;

```

```

12.  return poa;
13. }

```

It is important to point out that this representation represents the formal description of the function, and not the code description. This representation is not complete and presents only the main points of this function.

5.3 DSSV *find_POA*() Modeling

This function accepts two parameters (adapter and activate) and send back a POA_ptr. Starting from this, we can build the Coupled Model ports: $X = \langle \text{adapter}, \text{activate}, \text{find_poa}() \rangle$. “adapter”, “activate” are parameter ports while “find_poa()” is a call port. To these ports, we add to new ports (poacontrol and children) which 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; $X = \langle \text{adapter}, \text{activate}, \text{children}, \text{poacontrol}, \text{find_poa}() \rangle$. 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: $Y = \langle \text{Exception}, \text{POA_ptr} \rangle$.

The Coupled Model input ports are binded with the ports of the first met Atomic Model.

Line 2 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 sequencement ports which 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 toward getDestroyed() and one input parameter port for the returned value.

Line 3 defines a boolean “check”. This is done in our modeling approach using an Atomic Model (AM2) which will have to pass this variable through its ports. AM3 is the Atomic Model corresponding to line 4 (“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 8 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.

5.4 Simulation and Results

The simulation we present here focus on the behaviour of the function following different values for the “if” statement tests. The goal is to validate our approach using strings as

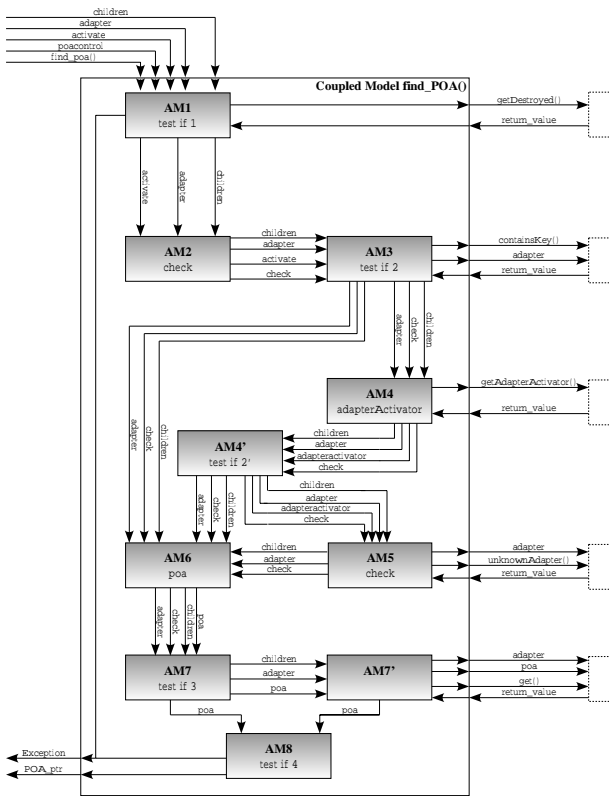


Figure 8: Modeling Scheme of the find_POA() Function

possible values for the various variables. Our approach is to consider some scenarios, to compute them “by hand”, to compute them using the DSSV simulation and then to perform a comparison between them. The scenarios we chose focus on atomic models sequencing paths. Figure 9 presents all these possible paths following the “if” test values.

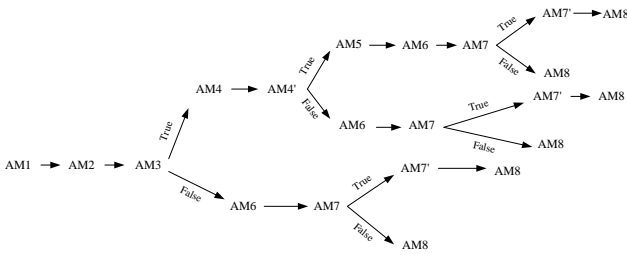


Figure 9: Possible Paths for Simulation

We found six scenarios presented in Table 1.

The Simulation has been performed using the JDEVS environment (?). This DEVS-based environment presents an easy-to-use graphical editor allowing a fast design of the model. Results of the simulation are showned in Table 2, and are very satisfying. All scenarios were well validated in a very short simulation time. We performed some others simu-

Scenario 1	AM3 test=true, AM4' test=true, AM7 test=true
Scenario 2	AM3 test=true, AM4' test=true, AM7 test=false
Scenario 3	AM3 test=true, AM4' test=false, AM7 test=true
Scenario 4	AM3 test=true, AM4' test=false, AM7 test=false
Scenario 5	AM3 test=false, AM7 test=true
Scenario 6	AM3 test=false, AM7 test=false

Table 1: The Six Scenarios for Simulation

lation (not presented here) and all of them gave good results.

Scenario	Simulation time	AM Real Path	AM Simulation path
1	0.5s	1, 2, 3, 4, 4' 5, 6, 7, 7', 8	1, 2, 3, 4, 4' 5, 6, 7, 7', 8
2	0.4s	1, 2, 3, 4, 4' 5, 6, 7, 8	1,2,3,4,4' 5, 6, 7, 8
3	0.4s	1, 2, 3, 4, 4' 6, 7, 7', 8	1,2,3,4,4' 6, 7, 7', 8
4	0.25s	1, 2, 3, 4, 4' 6, 7, 8	1,2,3,4,4' 6, 7, 8
5	0.15s	1, 2, 3, 6, 7, 7', 8	1, 2, 3, 6, 7, 7', 8
6	0.05s	1, 2, 3, 6, 7, 8	1, 2, 3, 6, 7, 8

Table 2: Simulation Results

6 CONCLUSION AND PERSPECTIVES OF WORK

We presented in this paper the DSSV methodology, an approach for the modeling and the simulation of distributed object architectures. We chose to present the find_POA() function which is one of the methods of the POA objects. This function is a good exemple for the methodology since it allows showing a great part of it. We can note that we performed the modeling and the simulation of the whole CORBA POA, and that we are currently working on the largest part of CORBA, said the ORB.

The DEVS-based approach for modeling and simulation is usually applied to physical systems. With DSSV, we propose a methodology that uses this approach for computer functions. This is possible thanks to the discrete-event nature of such systems. We think that coupling this methodology with a physical network simulation tool can allow to model a complete distributed object architecture over a network.

The main inconvenient of our approach can appear to be the building of big coupled models for a simple function. In fact, this is not a real problem since we use some graphical tools providing us a fast implementation. another point is that the atomic models are numerous, but also usually quite simple.

We have two main perspectives of work. The first one is to extend the DSSV methodology in order to include the latest development of the DEVS methodology. These developments should allow us to introduce in DSSV paradigms like dynamic and parallel modeling and simulation. With these

notions, DSSV should become more generic and should be used in more large-scale development than it is possible today. The second one, and probably the most important, is to perform a coupling with a low-level network simulation tool in order to simulate distributed object architectures over a network.

REFERENCES

- 2001a. The common object request broker architecture and specification.
- 2001b, December. Specification of the portable object adapter (poa).
- Aiello, A. 1997. *Environnement orienté objet de modélisation et de simulation à événements discrets de systèmes complexes*. Ph. D. thesis, University of Corsica.
- Booch, G., J. Rumbaugh, and I. Jacobson. 1998. *The unified modeling language user guide*. Addison-Wesley.
- Geib, J., C. Gransart, and P. Merle. 1999. *Corba: Des concepts à la pratique*. Dunod. 2nde Edition.
- Muller, P., and N. Gaertner. 2000. *Modélisation objet avec uml, deuxième édition*. Eyrolles.
- Pope, A. 1997. *The corba reference guide: understanding the common object request broker architecture*. Addison Wesley.
- Popper, J. 1973. *La dynamique des systhmes, principes et applications*. Les Editions d'Organisation.
- Pyarali, I., and D. Schmidt. 1998. An overview of the corba portable object adapter. Technical report.
- Schmidt, D., and S. Vinoski. Object interconnections, object adapters: Concept and technology. In *Proceedings*.
- Schmidt, D., and S. Vinoski. 1997. Object adapter: Concepts and terminology.
- Sommerville, I. 2001. *Software engineering*. Addison Wesley, 6th Edition.
- Zeigler, B. 1976. *Theory of modeling and simulation*. Academic Press.
- Zeigler, B., H. Praehofer, and T. Kim. 2000. *Theory of the modeling and simulation, 2nde édition*. Academic Press.

AUTHOR BIOGRAPHIES

EMMANUELLE de GENTILI received his master degree from the University of Corsica in 1998. She is currently pursuing the PhD degree in the SPE Laboratory, University of Corsica, France. Her current research interests relate to the theory of modeling and simulation, the DEVS formalism and the Software Validation and Testing using a generic method. This methodology is applied within a contract with Alcatel Telecom. Research Center on distributed systems. She is a SCS and IEEE member. Her e-mail address is gentili@univ-corse.fr.

FABRICE BERNARDI received his master degree from the University of Corsica in 1999. He is currently pursuing the PhD degree in the SPE Laboratory, University of Corsica, France. His current research interests relate to the theory of modeling and simulation, the DEVS formalism and the Object-Oriented Database Management Systems. He is a SCS and IEEE associate member. His e-mail address is bernardi@univ-corse.fr.

JEAN-FRANÇOIS SANTUCCI obtained his PhD in March 1989 at the University of Aix -Marseille - topic: a knowledge based system for testing of digital system. He has been Associated Professor at the University of Nimes from 1989 to 1995 and from 1995 to 1996 at the University of Corsica. He obtained his Professor ship in October 1995 at the University of Corsica. He is Professor in Computer Sciences since 1996 at the University of Corsica. His research interests are Modeling and Simulation of complex systems and High Level Digital Testing. He has been scientific responsible for several industrial contacts: european ESPRIT, Project EVEREST (European Vanguard Efforts on Research and Engineering of Systems for Testing) funded by the EEC, a research contact funded by the EOARD (european Office of Air Force Research and Development), responsible for the European Network BEL-SIGN (Behavioural design and test methodologies of digital systems) from 1995 to 1998. He his since 1998 Adjunct Director of the CNRS Research Laboratory UMR CNRS 6134, University of Corsica. His e-mail address is santucci@univ-corse.fr.