ENABLING ADVANCED SIMULATION SCENARIOS WITH NEW SOFTWARE ENGINEERING TECHNIQUES

Judicaël Ribault[a], **Olivier Dalle**[b]

[a][b]**MASCOTTE** project-team, INRIA Sophia Antipolis &
I3S, Université de Nice-Sophia Antipolis, CNRS
B.P. 93, F-06902 Sophia Antipolis Cedex, **FRANCE**.

[a]Judicael.Ribault@sophia.inria.fr, [b]Olivier.Dalle@sophia.inria.fr

ABSTRACT

In this paper, we introduce new techniques in the field of simulation to help in the process of building advanced simulation scenarios using preexisting simulation components. The first technique consists in using the Aspect Oriented Programming paradigm to capture some of the private data of an existing model component.The second one is an Architecture Description Language (ADL) designed for the Fractal component model, that offers definition overloading and extension mechanisms similar to those found in traditional Object Oriented languages. The benefits of using both techniques are illustrated by simple use cases of network security studies.

1 INTRODUCTION

In the 70's, Zeigler introduced the DEVS formalism (Zeigler 1976): a formalism to represent the hierarchical structure and behavior of discrete-event systems according to the Systems Theory. Later, Zeigler et al. further introduced in their Framework for Modelling & Simulation the concept of Experimental Framework (Zeigler, Kim, and Praehofer 2000). This Experimental Framework separates the computer simulation concerns in two parts: on one hand the model of the System Under Testing (SUT) and on the other hand, the Experimental Frame (EF). Hereafter, we will refer to the part of the Experimental Frame that generates exogenous events (inputs) for the model part, as the scenario part. This approach of separating concerns has benefits, such as allowing a better reusability of components.

From a methodological point of view, reuse allows to: (i) build reference model used in several studies, particularly to compare different solutions and (ii) benefit from user feed-back and/or improvements. Notice there are also situations in which reuse can simply not be avoided. Indeed, we may distinguish two levels of component availability. At source level, reusing an existing code offers enough flexibility to allow any desired modification (but at the cost of losing the results of a previous verification and validation.) On the contrary, when components are only available in compiled object code, reuse necessarily happens without any modification.

Furthermore, the approach of separating concerns may imply some limitations. For example, Systems Theory normally prohibits direct interactions between the scenario part and the inner parts of models, because interactions have first to go through the boundaries of the outer components of the model in order to reach the inner ones. Furthermore, for some studies, it may be useful to extend the previous definition of a scenario to include, in addition to the ability to send exogenous events to the model, the ability of applying structural changes to an existing model (before the simulation starts running).

In this paper, we describe new techniques coming from the field of software engineering that can be used in the field of simulation to get around these limitations while enforcing the separation of concerns principles of the Experimental Framework. Hence, it is worth noting that separating models and scenario allows a better reuse of components in both parts: reuse of a given model with various scenarios, or reuse of a given scenario with various models. In particular, it is often advocated that a model that can be reused multiple times or used in combination with other models can save a many time, money, and human effort (Davis and Anderson 2003).

Section 2 present the software background involved in this paper. Section 3 present the use case in which we present the use of ADL (section 3.2) and AOP (section 3.3).

2 BACKGROUND

This section first introduce the Open Simulation Architecture (OSA) (Dalle 2007), a discrete-event simulator that provides a process-oriented programming model. Then we put the emphasis on the two techniques used for its implementation that are of particular interest for building advanced scenarios.

2.1 Open Simulation Architecture

The goal of OSA is to help users in their simulation activities like building models, developing simulations campaigns, running experiences plans, or analyzing data results. Also, OSA aims at becoming framework for the modelling and simulation community by favoring the integration of new or existing contributions at all levels architecture. Figure 1 represents the OSA architecture. In the left part, the front end-users GUI based on Eclipse framework. In the center part, the functional concerns and in the right part the simulation tasks. Functional concerns resolve one or more typical simulation tasks. Each functional concerns are part of the OSA software components and must be considered optional and replaceable independently from one another. In OSA, handling are almost always hidden in the controller component thus significantly reduce the modelling process, but also simplifies the replacement of any part of the simulation engine. OSA allows to model component-based systems using Fractal component (Bruneton, Coupaye, and Stefani 2004). AOKell, an open implementation in Java of the Fractal component model, provides an aspect-oriented approach to integrate control concerns in component. In practice, the real system is represented by a FractalADL application. This application can then be instrumented using Fractal component capability.
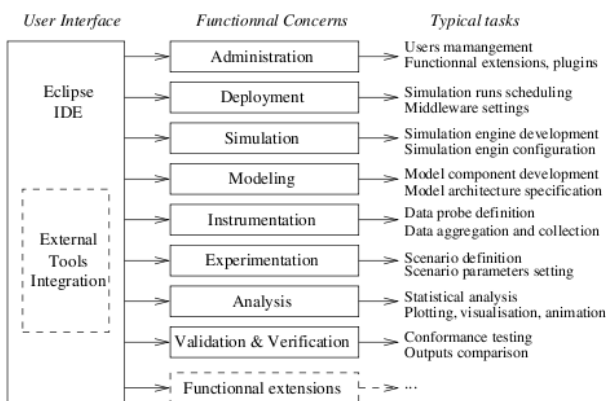

Figure 1: OSA functional architecture.

2.2 Fractal component

Fractal basis development lies in writing components and connections that enable components communication. Fractal specification is based on: (1) hierarchical components that provide a uniform view of applications at different levels of abstraction, (2) shared components that allow modelling and sharing of resources, while preserving hierarchical components, (3) introspection to observe the performance of a system, and (4) (re)configuration capabilities that enable deployment and dynamic system configuration. Furthermore, Fractal is an extensible model because it allows the developer to customize the control capabilities of each application's component. A Fractal component is an unit of deployment that have one or more interfaces. An interface is an entry point to the component. An interface implements an interface type,

which specifies the operations supported by the interface. There are two types of interfaces: server interfaces that correspond to the services provided by the component and client interfaces that correspond to services required by the component. A Fractal component is normally composed of two parts: a membrane which possesses functional interfaces and interfaces allowing introspection and (dynamic) configuration of a component, and a content that is made up of a finite set of sub-components.

Figure 2 shows an example of Fractal component. Components are represented by rectangles. The bold line corresponds to the membrane component. The inner part corresponds to the content of the component. Interfaces are represented by round for clients interfaces, and by empty half-round for servers interfaces. Note that internal interfaces allow a hierarchic component to control the exposure of its external interfaces to its sub-components. External interfaces appearing at the top of the components are component control interfaces. dashed line represent connections among components. Fractal provide a Architecture Description Language (ADL) (Clements 1996; Medvidovic and Taylor 2000) to describe applications architecture.
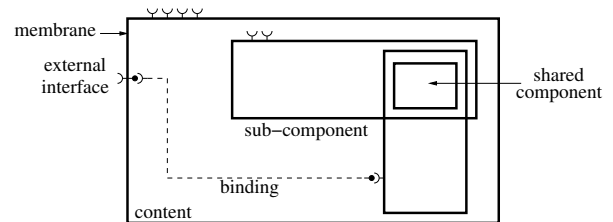

Figure 2: Fractal component example.

2.2.1 Fractal ADL

FractalADL is a XML language to describe the architecture of a Fractal application: components topology (or hierarchy), relationship between client and server, name and initial value of components attributes. A FractalADL definition can be divided into several subs definitions and several files. Moreover, the language supports a mechanism to ease the extension and redefinition through inheritance. The motivation for such scalability is twofold. On the other hand, the component model itself is extensible, it is possible to attach an arbitrary number of components controllers. There are multiple uses for a given ADL definition: deployment, verification, analysis, and so on. FractalADL allows to separate concerns because model definition can be split in multiple files. ADL language is interpreted by a specialized component of Fractal called a Factory: to read completely (recursively) a description of a Fractal application, just send a request to the Fractal Factory to read and instantiate the root component of the application. To instantiate the various components, the factory creates a Fractal Abstract Syntax Tree (AST), where each node corresponds to a XML entity of the ADL.

2.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) (Elrad, Filman, and Bader 2001) is a new paradigm for modularizing applications with many concerns. AOP goals are (i) separation of concerns: the goal is to design systems so that functions can work independently of other functions, and so it is easier to understand, design and manage complex interdependent systems; (2) crosscutting interactions: it is not easy to modularized common-interest concerns used by several modules, like logging service; (3) dependencies inversion: instead module use well-known services, the well-know service shall use modules.
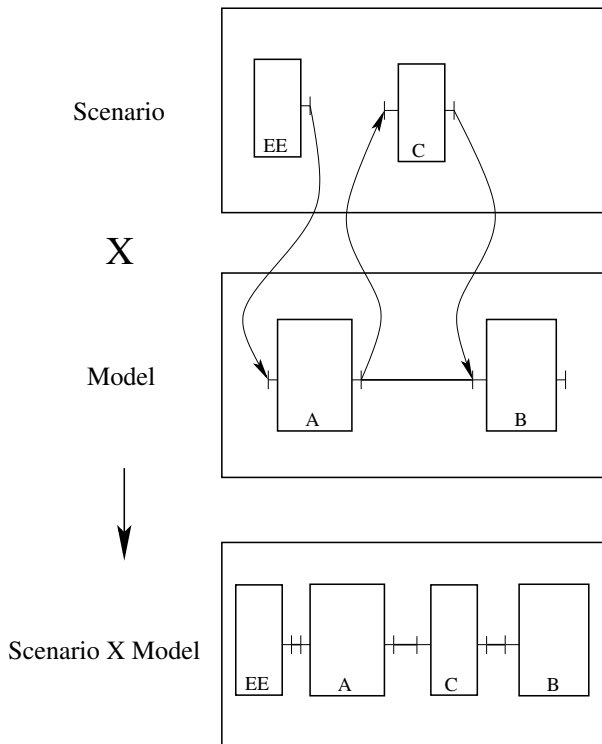
3 ADVANCED SCENARIOS CASE STUDIES

Figure 3: Reuse and adapt a model of reference.

We present hereafter two case studies to illustrate the new techniques we propose to build advanced scenarios reusing existing component models that are only available in compiled form, at execution level (for example because it came after a long validation and verification process, or because we want to keep the source code secret). Figure 3 show the composition of the complex scenario and the reference model. The reference model contains two components A and B. The complex scenario adds a new component C between A and B, and a new component EE which generates exogenous events. The composition is the result of the model and the scenario. In order to build such a composition we propose to use (i) an Architecture Description Language (ADL) with overloading capability like FractalADL and (ii) Aspect-Oriented Programming (AOP) like AspectJ. to extend the reusability of a model through a simple case study To illustrate this kind of composition we build

a practical example: a small security case study based on a reference model in which a user establishes an FTP session with a server using the unsecured version of the protocol. The case study will consist in simulating a Man-In-The-Middle attack (MITM) and a Spy-ware version of the client.
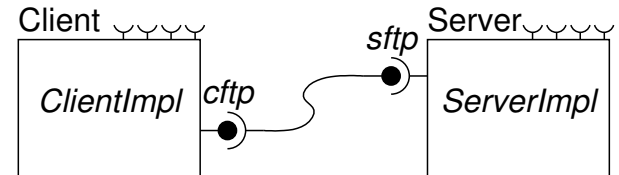
3.1 Case study

Figure 4: Components layout of File Transfers Protocol case study.

As described previously, we propose to use AOP and ADL in an original way to override difficulties in reusing models. We choose in this paper to show the cost and benefits through a simple case study. First, let us assume that we have a model we want to reuse to test different security flaws. There is a model representing the Basic operation of a server File Transfer Protocol (FTP). This simple model has not been developed in order to be used in this study, we are not supposed to have the source code, and even we need to test the safety of this protocol. Figure 4 shows the architecture of the model, and Listing 1 details its implementation in FractalADL. Line 4 specifies the name of this model, line 6-11 correspond to the client definition and line 12-19 to the server definition. Line 7-9 and 14-16 describe client and server interfaces used by the binding on line 20-21.

Listing 1 Fractal ADL definition used to implement layout of figure 4.

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition skipped ... >
03
04<definition name="ftp">
05
06  <component name="Client">
07    <interface name="cftp"
08              role="client"
09              signature="FTPService"/>
10    <content class="ClientImpl"/>
11  </component>
12
12  <component name="Server">
14    <interface name="sftp"
15              role="server"
16              signature="FTPService"/>
17    <content class="ServerImpl"/>
18  </component>
19
20  <binding client="Client.cftp"
21           server="Server.sftp"/>
22</definition>
```

The protocol represented by this model is a two-party protocol. We will denote the two parties by the name Client and Server (Client want to be authenticated on Server). The model works like this : the client send the users login and password to the server to be authenticated. To do this, client ask his interface (cftp, declared line 07) to obtain connection with the server. In this study, we focus on the login process to test security flaw.

From this model, we propose a new reusing approach. First, we will show how to add a man in the middle attacker in this model using the overload capability of FractalADL. Second, we will show how to simulate spy-ware on client using the overload capability of FractalADL and AOP.

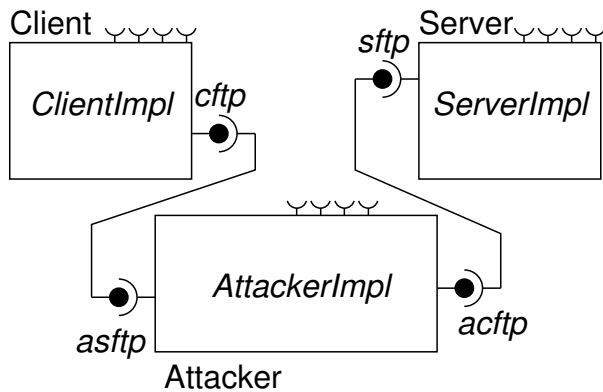## 3.2 Man-in-the-middle attacker with Fractal ADL

Figure 5: Components layout of Fractal's MITM attack.

From the original model describe in section 3.1, we want to test the ftp login process security. We decide to test the security against a man-in-the-middle attacker. In the man-in-the-middle setting (MITM), there is a third party called Adversary. All the communication between Client and Server are intercepted by Adversary. Thus both Client and Server talk to Adversary and cannot communicate directly with each other. Adversary need to transmit information between Client and Server, but - it's the security break - he can read, change, or drop transmit depending on his settings.

What makes this case interesting is to modify the original FTP topology (figure 4) to obtain the new topology describe in figure 5. In practice, we need to add a new component inside a model. Like in reality, Adversary need to mimic Server interface and Client Interface. In fact, Adversary need to imitates Server for the Client, and imitates Client for the Server. Figure 5 show the new architecture we want to obtain compared to figure 4 section 3.1. Since model is locked, we cannot change his topology directly in source code. Listing 2 shows how to use the FractalADL overload capability to overload the topology. Line 04 show we extend the original ftp model in a new model called mitm-ftp. Line 06-14 represent the declaration of the new Adversary component. And line 16-19 demonstrate how overload the original

Listing 2 Fractal ADL definition used to implement layout of figure 5.

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition skipped ... >
03
04<definition name="mitm-ftp" extends="ftp">
05
06  <component name="Adversary">
07    <interface name="acftp"
08                role="client"
09                signature="FTPService"/>
10    <interface name="asftp"
11                role="server"
12                signature="FTPService"/>
13    <content class="AdversaryImpl"/>
14  </component>
15
16  <binding client="Client.cftp"
17            server="Adversary.asftp"/>
18  <binding client="Adversary.acftp"
19            server="Server.sftp"/>
20</definition>
```

binding between Client and Server by a new binding between Client and Adversary, and between Adversary and Server. With this topology, communication between the Client and the Server pass through the Adversary.

This example shows how to modify a model to include new component or change topology. The overload capability of Fractal ADL permit to reuse and change some specification of the model like topology. In fact, in our example, communication between the Client and the Server go through the Adversary but the FTP model have not been modified. We build a new model extending the original FTP model, and overload the binding between the Client and the Server. In the next section, we use FractalADL to add a new component and change the topology, but we also demonstrate how to use AOP. The next section described the FTP model with a spy-ware inside the client.

3.3 Spy-ware with aspect-oriented programming
In this section, we demonstrate how using Fractal ADL and aspect-oriented programming we can add a spy-ware (Stafford and Urbaczewski 2004) into the Client from the original FTP model. Spy-ware is the name given to the class of software that is surreptitiously installed on a computer and monitors users activities and reports back to a third party on that behavior [Anon, 2004; Daniels, 2004; Doyle, 2003; Taylor, 2002]. We want to model a spy-ware inside the Client of the FTP model. The goal of this attack is to take the user login and password when typed in. Spy-ware send all information to a third party using the network. The model architecture we want to obtain is shown in figure 6. We see the Client is connected to a third entity (Spy) and contain a Spy-Ware inside his implementation.

Listing 3 shows a solution using Fractal ADL and AOP to introduce spy-ware in original FTP model. Using the extension capability of Fractal ADL, we add a new spy interface to the Client component, we add a Spy component and we bind the Client and
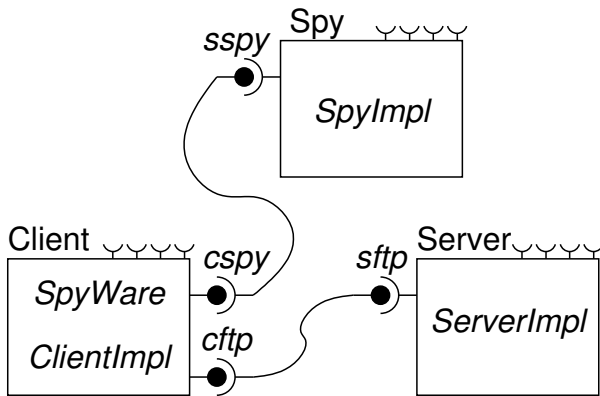
Figure 6: FTP model with Spy-Ware in Client.

the Spy together. Line 04 show how to create a new model extending the original FTP model. Line 06-17 represent the Spy component, line 07-09 represent the interface for connecting with the Spy component. Line 13-17 represent the Client component declared in the original FTP model, line 14-16 show the new interface added to the Client component. Line 19-20 represent the binding to connect the Client with the Spy component.

Listing 3 Fractal ADL used to implement layout of figure 6.

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition skipped ... >
03
04<definition name="spyware-ftp" extends="ftp">
05
06  <component name="Spy">
07   <interface name="sspy"
08              role="server"
09              signature="SpyService"/>
10   <content class="SpyImpl"/>
11  </component>
12
13  <component name="Client">
14   <interface name="cspy"
15              role="client"
16              signature="SpyService"/>
17  </component>
18
19  <binding client="Client.cspy"
20           server="Spy.sspy"/>
21</definition>
```

AOP allows us to introduce new code into objects without the objects is needing to have any knowledge of that introduction. The FTP model has been validated and we don't have the source code so we can't change it to introduce some concerns about spy-ware. The Listing 4 show how using AOP we can add some concerns inside a model.

Line 01 explain we want to intercept a method call, and do something before the method was called. Line 02 show the method we want to intercept, it's all methods from the FTPService java interface called by a ClientImpl class. Line 03 add a condition, only component bound with a Spy component are concerned. Line 05 ask the Client interface connected to the Spy component to have

this one. Line 06 call through the connection with the Spy the send method to send data. This aspect (written in AspectJ) represent the Spyware, the Spy component represent the third party waiting for data to analyze.

Listing 4 Fractal ADL used to implement layout of figure 6.

```
01 before(ClientImpl b) :
02   call(* FTPService.*(..)) && this(b)
03   && if(isBinding(b)) {
04   try {
05     SpyService spyS = b.lookupFc("cspy");
06     spyS.send(thisJoinPoint.getArgs()[0]+"");
07   } catch (NoSuchInterfaceException nsie) {
08     ...
09   }
10 }
```

This example shows how to modify a model to include new component, change topology and instrument a component. The capability of AOP to inject some code inside the model allow to read variables of the model. Here we demonstrate how a third component can access the login and password field during the login process of the client on server.

4 CONCLUSION

We have shown how ADL and AOP techniques can be used to extend the reusability of a model. Both techniques offer new ways to create a complex scenario without modifying the original model. Hence, the model remains valid which saves additional costs and efforts. The ADL allows to build a composition of the model with the scenario by overloading some model definitions like bindings. AOP helps to add some code into the model, for example to allow a third party component to access an existing model's private data. However, this latter technique must be used with extreme care in order to guarantee that the code newly inserted in a component will not change its behavior. However, tools can be built to make automatic verifications on the code inserted and ensure this non-interference property. Our planned future works are to build a new DEVS-compliant engine for OSA in order to experiment these techniques on existing DEVS models. However, we want also to further investigate the benefits and drawbacks of using ADL and AOP techniques with the DEVS Modelling & Simulation framework. A last direction we want to explore is the identification of practical use cases in which such techniques prove to be useful, in particular in the networking and security area, where models and scenarios exhibit a priori a high complexity(Seo 2006).

REFERENCES

Bruneton, E., Coupaye, T., and Stefani, J. B.. 2004, February. The fractal component model specification. Available from http://fractal.objectweb.org/specification/. Draft version 2.0-3.

Clements, P. C. 1996. A survey of architecture description languages. 16: IEEE Computer Society.

Dalle, O. 2007, Februray. Component-based discrete event simulation using the Fractal component model. In *AI, Simulation and Planning in High Autonomy Systems (AIS)-Conceptual Modeling and Simulation (CMS) Joint Conference*. Buenos Aires, AR.

Davis, K. P., and Anderson, A. R.. 2003. *Improving the composability of department of defense models and simulations*. RAND Technical report available at http://www.rand.org/publications/MG/MG101/ (last accessed April 2008).

Elrad, T., Filman, R. E., and Bader, A.. 2001. Aspect-oriented programming: Introduction. *Commun. ACM* 44 (10): 29–32.

Medvidovic, N., and Taylor, R. N.. 2000. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.* 26:70–93.

Seo, H. S. 2006, July. Network security agent DEVS simulation modeling. *Simulation Modelling Practice and Theory* 14 (5). doi:10.1016/j.simpat.2005.08.010.

Stafford, T. F., and Urbaczewski, A.. 2004. Spyware: The ghost in the machine. 291–306: Commun. AIS 14.

Zeigler, B. P. 1976. *Theory of Modelling and Simulation*. Wiley & Sons, NY.

Zeigler, B. P., Kim, T. G., and Praehofer, H.. 2000. *Theory of Modeling and Simulation*. Academic Press, Inc.

AUTHOR BIOGRAPHIES

JUDICAEL RIBAULT received his M.Eng. from University of Nice-Sophia Antipolis in 2007. In 2007, he joined the MASCOTTE common project-team of the I3S-UNS/CNRS Laboratory and INRIA, in Sophia Antipolis, where he started to work on the OSA project as a Software Engineer. In 2008, he started a PhD in the MASCOTTE team, in the context of the ANR SPREADS project, investigating on new Component-based Software Engineering techniques for large scale Peer-to-peer systems simulation.

OLIVIER DALLE is associate professor in the C.S. dept. of Faculty of Sciences at University of Nice-Sophia Antipolis (UNS). He received his BS from U. of Bordeaux 1 and his M.Sc. and Ph.D. from UNS. From 1999 to 2000 he was a post-doctoral fellow at the the French space agency center in Toulouse (CNES-CST), where he started working on component-based discrete event simulation of complex telecommunication systems. In 2000, he joined the MASCOTTE common project-team of the I3S-UNS/CNRS Laboratory and INRIA, in Sophia Antipolis.