AN INSTRUMENTATION FRAMEWORK FOR COMPONENT-BASED SIMULATIONS BASED ON THE SEPARATION OF CONCERNS PARADIGM

Olivier Dalle, Cyrine Mrabet

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

Olivier.Dalle@sophia.inria.fr (Olivier Dalle)

Abstract

This paper presents the authors' ongoing work in applying the "Separation of Concerns" (SoC) software design paradigm to the Instrumentation Framework of a new component-based simulation platform called OSA (Open Simulation Architecture). The SoC paradigm emerged recently from the research in Component Based Software Engineering (CBSE). It consists in enforcing the strict separation of the instructions of a program as soon as these instructions have different functional goals. The first expected benefit of this approach is to improve the re-usability of the resulting software. In our particular case, applying the SoC paradigm to the Modeling and Instrumentation concerns means that several instrumentations can be developed for a given model without any change to the modeling code. Following, the same modeling code may be reused in various computer simulations and with possibly very different instrumentations, depending on the goals of the study. The second expected benefit of applying the SoC approach to the Instrumentation Framework of OSA, as shown in this paper, is to gain a better control on the instrumentation overhead.

Keywords: Instrumentation, **Framework**, **Discrete-Event Simulation**, **Component Based Software Engineering**, **Aspect Oriented Programming**.

Presenting Author's Biography

OLIVIER DALLE is *Maître de Conférences* (Associate Professor) in the C.S. dept. of Faculty of Sciences at University of Nice-Sophia Antipolis (UNSA). He received is BS from U. of Bordeaux 1 and his M.Sc. and Ph.D. from UNSA. 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 project, a common team of the I3S-UNSA/CNRS Laboratory and the INRIA Research Unit, in Sophia Antipolis. His web page can be found here: `http://www.inria.fr/mascotte/Olivier.Dalle/`.

ISBN 978-3-901608-32-2

1 Introduction

A very common motivation for reproducing the behavior of a system in a computer simulation is to be able to virtually *observe* (collect data about) the behavior of that system. The Instrumentation Framework (IF) refers to the software pieces and programming instructions required in such computer simulations in order to implement observations. Computer simulation programs usually mix a generic (reusable) technical part, often referred to as the *simulation kernel*, and more specific parts that depend on the system under study, usually simply referred to as the *model implementations* or *modeling code*.

In many simulators, the IF is deeply interleaved in both parts. When no or little instrumentation support is provided by the simulator environment (within the kernel, or thanks to dedicated libraries), the coding part of the instrumentation is left up to the model developer.

As an example, and without entering yet in the details, simply consider the proportion of instructions that appear in bold font in Listing 1: these bold instructions represent the instrumentation part while the normal ones represent the modeling part of the example. Clearly, in this example, the instrumentation represents a significant part of the code. As will be further discussed in section 2, no effort was done in this example to minimize the set of instructions used for instrumentation; but in any case, without special programming techniques such as the one later described in this paper, it is nearly impossible to avoid mixing of modeling and instrumentation instructions.

Listing 1 An instrumented model (Java).

```
class MobileObject {
  private float X;
  private float Y;
  private SimTime last_time;
  private Sampler PosSampler =
       new Sampler("POS");

  public void newpos(float dX, float dY) {
    int sample = 0;
    X = X + dX * (Now() - last_time);
    Y = Y + dY * (Now() - last_time);
    sample = sample +1;
    PosSampler.write("X"+sample+"="+X);
    PosSampler.write("Y"+sample+"="+Y);
  }
...
```

Indeed, instrumentation instructions typically start, at the lowest level of the instrumentation, by collecting raw data from the model. Since the model developper implements the model, he/she is in the best position for implementing this collection. For this purpose, and depending on the services and libraries provided by the simulator being used, he/she may use existing libraries or services for generating traces, or implement his own trace generation or data collection mechanism. For example, a common practice is to offer support in the simulation API for the declaration of *observable objects* within the modeling code.

In any case, these common practices imply that whatever the goals of the study are, all the possible observations for a given model need to be decided (and hard-coded) at the time the model is implemented. As a consequence, two strategies may be envisaged: a lazy one in which the model developer only provides a minimalistic set of observable objects, or an exhaustive one in which the model developer tries to identify the exhaustive set of potentially useful observable objects.

In terms of reuse and performance both approaches have opposite benefits and drawbacks: assuming that the execution overhead of the observation is related to the cardinality of the set of observable objects, then we may expect a better performance from the lazy approach than from the exhaustive one; but when trying to reuse the modeling code for another study, the risk to not find the needed objects in the set of observable objects is higher with the lazy approach, which means that there is a high probability that the code will need to be modified. And modifying the code of an existing model simply for instrumentation purposes is a questionable practice. For example it raises the problem of supporting potentially conflicting instrumentations released by un-coordinated parties or simply the problem of properly switching from one instrumentation to another in a simulation study where the same model is reused many times in various contexts.

In this paper we describe a new approach that gains the benefits of the two previous approaches without paying for their drawbacks. This approach is based on the Separation of Concerns (SoC) paradigm[1], which is a fundamental basement in recent software designs and related programming techniques, such as Component-based Software Engineering (CBSE) and Aspect Oriented Programming (AOP)[2, 3].

As its name suggests, applying the SoC paradigm in our case consists in separating both the modeling and instrumentation concerns from the very beginning of the simulator design, such that the instructions that serve each concern are *never* found intermixed. However, the instrumentation instructions are still strongly connected to the modeling instructions, since the primer performs computations on the latter ones. On the contrary, the modeling instruction should be kept totally independent from the instrumentation ones.

2 A Simple Case Study

Let's consider again, in more details, the Java code of Listing 1. In this example, we assume that the modeling part of the code simply consists in recomputing the position of a mobile object according to it's current speed in the plan (`dX` and `dY`) , it's previous position (`X` and `Y`) and the elapsed simulated time since last update. The instrumentation part of this code consists in generating traces of the successive values of the two variables `X` and `Y` in a sampling output channel tagged with label `"POS"`.

This example illustrates how the instrumenting part of the code (emphasized in bold font) can easily represent

ISBN 978-3-901608-32-2

a noticeable part of the overall code. Indeed, these bold instructions reflect the typical set of instructions needed for building an instrumentation:

1. Get successive values of a variable (or object) of the model. These values are usually called the *samples* and the sequence of values a sample-path[4];
2. Associate a different time-stamp or sequence number to each of these samples;
3. Forward the samples for further on-line processing (trace generation, statistical processing, logging facility, threshold detection, etc.)

These needs may either have to be explicitly implemented, as in our example, or implicitly provided by a language construction. In the latter case, the general technique consists in ensuring that all the observed objects, such as `X` and `Y` in our previous example, share a common set of (reusable) services for sampling. A possible application of this construction is the code of Listing 2. In this second implementation of the `MobileObject` class, the sampled variables `X` and `Y` are no more primitive types (`float`), but a complex type whose definition may resemble to that of the class `SmplFloat` of Listing 3.

Notice that thanks to this second implementation, the instrumentation "footprint", or in other words, the size of instruction set used for the instrumentation is significantly reduced in Listing 2 compared to the previous version of Listing 1. Notice also that the instructions found in Listing 1 are still needed, but are now split in two parts, one reusable in class `SmplFloat` and one specific, in class `MobileObject`.

Listing 2 Same model as in Listing 1 with a smaller instrumentation "footprint".

```
class MobileObject {
  private SmplFloat X = new SmplFloat("POS");
  private SmplFloat Y = new SmplFloat("POS");
  private SimTime last_time;

  public void newpos(float dX, float dY) {
    X.set(X.get() + dX * (Now() - last_time));
    Y.set(Y.get() + dY * (Now() - last_time));
  }
...
```

Even a reduced instrumentation "footprint" such as the one presented in Listing 2 seriously impedes the reusability of the model code for other experiments. Indeed, if the simulationist is interested in producing other samples than the one that are *hard-coded* in the model, then the source-code of the model needs to be modified. Notice also how, in our particular example, the change from a primitive (`float`) type to a complex type directly impacts the modeling code (affectations and setting of `X` and `Y` are replaced by setter/getter functions), due to some Java language technical constraints.

Listing 3 Definition of the `float` sampling object used in Listing 2.

```
class SmplFloat {
  private int sample;
  private float val;
  private String tag;
  private Sampler sampler;

  public SmplFloat(String smpl, String tag) {
    sampler = new Sampler(smpl);
    this.tag = tag;
  }

  void set(float new_val) {
    val = new_val;
    sample = sample +1;
    sampler.write(tag+sample+"="+new_val);
  }
...
```

The goal of the Instrumentation Framework presented in this paper is to allow any instrumentation on a model with the guarantee of a *void* "footprint". In other word, we want to be able to produce the same results as the one produced in the example of Listing 1 but with the constraint that the code of the model never needs to be modified and contains no other instructions than the ones of Listing 4. Of course, the instrumentation instructions have to be placed somewhere, but our goal is to ensure that these instructions are never found interleaved with modeling instructions.

Listing 4 The goal: same model as in Listing 1 with a *void* instrumentation "footprint".

```
class MobileObject {
  private float X;
  private float Y;
  private SimTime last_time;

  public void newpos(float dX, float dY) {
    X = X + dX * (Now() - last_time);
    Y = Y + dY * (Now() - last_time);
  }
...
```

3 Overview of OSA

OSA (Open Simulation Architecture) is a new collaborative platform for component-based discrete event simulations[5, 6, 7]. It relies on the ObjectWeb's Fractal component model[8] and one its Java-based implementation called AOKell[9]. The front-end Graphical Interface is based on the Eclipse IDE[10, 11]. OSA also provides a public repository based on the Apache Foundation's *maven* building system that automatically computes the dependencies between the components used for a given simulation.

3.1 The Fractal Component Model

Fractal is the ObjectWeb Consortium component reference model. Fractal is *neither* a software environment *nor* a run-time executive. It is a specification. In other words, it is a set of rules and features that a component-

based software architecture is supposed to follow or implement in order to be compliant with this model. Fractal does not mandate the use of any specific programming language. On the contrary, it allows to combine component implementations possibly based on different programming languages.

Hereafter, we summarize some of these key features (see [12] for the complete description).

Component external structure

A Fractal component is an object-oriented unit of code that has external interfaces. These interfaces may be of two kinds: either *client* or *server*. The former emits service requests, the latter receives service requests. Interfaces are named. Their name must be unique for a given component but names may be reused for naming interfaces in other components. A client interface is intended to be bound to a server interface.

Hierarchical structure

Components may have a hierarchical structure (fig. 1). Hierarchical components are made of a controller part (also called membrane) and a content part. The content part is composed of one or more components. Since a controller and its content recursively form a component it may have external interfaces. It may also have internal interfaces. As external interfaces, internal interfaces may be either of type client, or of type server. Internal interfaces are only available to components of the content part. A component of the inner part may only bind its external interfaces to external interfaces of other inner components or to the inner interface of its surrounding controller. Therefore, the model strictly forbids a component to bind its external interfaces to the ones of components outside its controller or inside its neighboring (inner part) components.

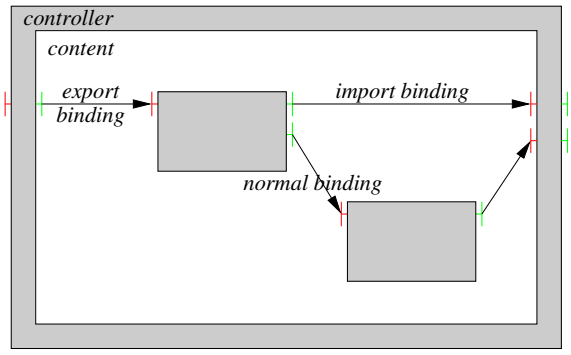

Fig. 1 Example of Fractal hierarchical component.

Introspection

Introspection is the ability for an object to collect useful information about other objects (possibly including itself). In the Fractal model, components have the ability to introspect their interfaces. For example, a component may retrieve its own list of available internal and external interfaces. Depending on the component type, they may also have other introspection capabilities, such as inner content architecture for composite (hierarchical) components.

Functional and controller interfaces

A functional interface is an interface used to offer or obtain services to or from other components. A controller interface is a server-only interface that is offered to the content part of a component to access non-functional services, such as introspection, (re)configuration, persistence, service policy, life cycle control (ability to start/stop a component), and so on.

Factories and templates

A factory component is a component that has the ability to create other components. Fractal distinguishes two kinds of factories: generic factories, that have the ability to create several kinds of components, and standard component factories, that only have the ability to create one kind of component. Templates components are a special kind of standard factory components, that may be recursively composed of factories, and serve as a model to create normal components in a quasi isomorphic manner (isomorphic meaning the created component has the same hierarchical structure as its creator template). Since factories are components and components are created from factories, a special component is required to initiate the recursion. This special component is a generic component factory called “bootstrap”.

Shared components

The Fractal model allows a component instance to appear simultaneously in the content of several distinct enclosing components. Such components are called *shared components*. This property has two noticeable consequences: (i) a shared component is placed under the control of several surrounding controller components and (ii) a shared component may directly interact with components located in the inner parts of several distinct components.

Architecture Description Language

Fractal’s Architecture Description Language (ADL) is a convenient mean for describing the architecture of the components that form a Fractal application. Fractal’s ADL is an XML based language that describes the topology of (hierarchical) components, client/server bindings, name and initial attribute values of components. An example of ADL file is given in Listing 5. The corresponding Fractal application is illustrated on figure 2.

The Fractal ADL definitions can be split in various files. Furthermore, the language supports extensions and provides an heritage mechanism to ease the overloading of definitions. Therefore, the Fractal ADL allows for a proper separation of concerns, despite no particular layout is enforced in the Fractal specification. The ADL language is parsed by a specialized Fractal factory component: in order to fully (recursively) read the description of a Fractal application, it is sufficient to ask this factory to read and instantiate the root component of the application which is no more than one line of Java code.

ISBN 978-3-901608-32-2

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

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition skipped ... >

<definition name="HelloWorld">
  <interface name="m" role="server" signature="java.lang.Runnable"/>

  <component name="Client">
    <interface name="r" role="server" signature="java.lang.Runnable"/>
    <interface name="cs" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>

  <component name="Server">
    <interface name="ss" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>

  <binding client="this.m" server="Client.r"/>
  <binding client="Client.cs" server="Server.ss"/>
</definition>
```

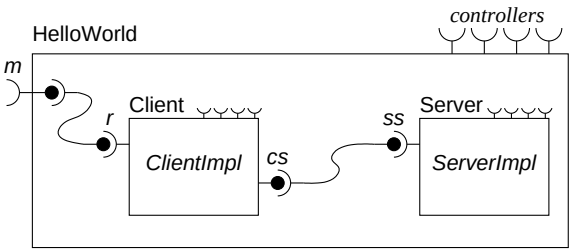

Fig. 2 Components layout of Fractal's HelloWorld example.

3.2 The OSA Architecture

The OSA software architecture follows a layered (or N-tier) design, in which the layers mainly correspond to the various functional concerns identified in fig. 3. Since one the strongest design principles of OSA is to separate concerns, each of these layers is designed independently and kept self-contained as much as possible. Eventually, these "concern-layers" are interleaved and mixed at the very last time, using AOP and CBSE techniques and tools such as AspectJ and Fractal-AOKell.

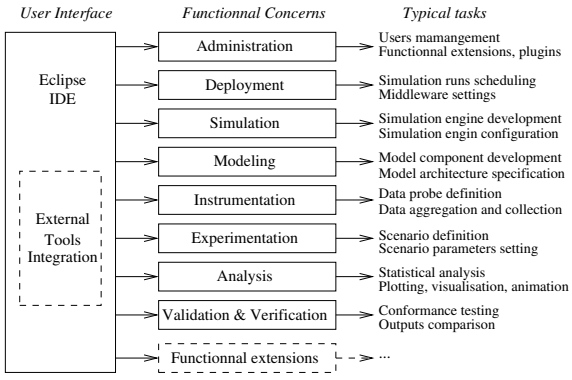

Fig. 3 OSA functional architecture.

User Interface Layer

The OSA architecture must provide tools to assist users in many tasks. Furthermore, the architecture should enforce a strong cooperation of these tools, using an integrated and easily extensible environment. For this purpose, as shown on fig. 4, we selected the Eclipse platform[10, 11] as the front-end Graphical User Interface.

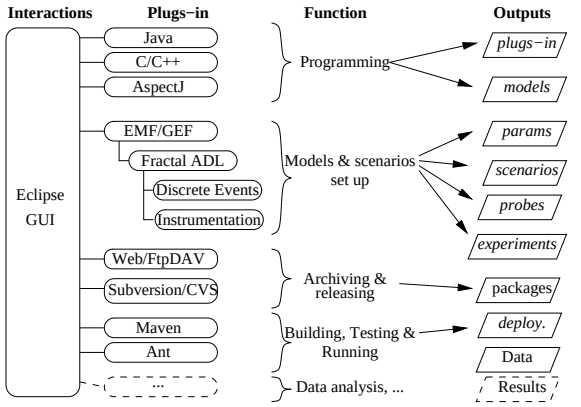

Fig. 4 OSA's Eclipse-based Graphical User Interface and Interactive Functions

Eclipse already provides a large amount of *plugs-in* to assist developers in various software development tasks: specification, development in several programming languages, unit testing, debugging, source code management, and so on. Some of these plugs-in are dedicated to the development of new Eclipse plugs-in, which explains the ever growing list of available plugs-in, and consequently its ever growing popularity.

An interesting feature of the Eclipse plugs-in is their ability to be extended. Indeed the Eclipse plug-in API defines *extensions points*[13]: plugs-in that implement such extension points (this is not mandatory) may be extended in order to build new enriched or specialized versions of the initial plugs-in.

ISBN 978-3-901608-32-2

Eclipse plugs-in are mainly used to build new Eclipse *perspectives*. An Eclipse perspective is dedicated organization of the Eclipse Graphical User Interface (GUI), offering support and specialized tools for a particular task.

Therefore, the development of the OSA user interface mainly consist in providing new Eclipse plugs-in and perspectives to support users in (possibly) all the tasks of the simulation study life-cycle. For example, on fig. 4, one of our contribution is the Fractal ADL plug-in, built on top of the Eclipse EMF and GMF libraries (Eclipse Modeling Framework and Graphical Editing Framework). This general purpose plug-in for editing Fractal ADL files offers extensions points for specialized ADL extensions, such as the ones we have implemented for the OSA needs (eg. for discrete event scheduling and instrumentation setting).

Execution Layer

A middle-ware layer may optionally be used to support the execution of the simulations. Indeed, such architectures are often criticized for their potentially poor performance. Since performance is a critical issue for simulation, this architectural choice may be questionable. As a matter of fact, the middle layer is not mandatory in the OSA architecture. Indeed, in the Fractal component model, the distribution of component executions across a network through a middle-ware is an optional feature that may, or may not be activated. Since this is implemented as a non functional feature of components, the decision of activating or not this feature is totally transparent. In other words, it does not require any change in the component functional implementation.

The OSA architecture allows the distribution of the simulation executions across the network in different manners:

- distribution of several simulation-runs, each one executing on a single computer node. In this case, the distribution support required is very limited (a "gang-scheduler" facility);
- distribution of one (or several) simulation runs across the network, simply using the Fractal model ability to distribute transparently the execution of the components, but *without any cooperation of the simulation engine*. Since the minimal requirement of the simulation engine, whether it executes in parallel or not, is to ensure consistency of event processing between components, this implements de facto the so-called *conservative* mode of parallel execution[14];
- distribution of one (or several) simulation runs across the network, using the Fractal model ability to distribute the execution of the components, and *the cooperation of the simulation engine*. Provided the components have the persistence non-functional feature in order to regularly save their global simulation state, this lead to the so-called *optimistic* mode of parallel execution[14];

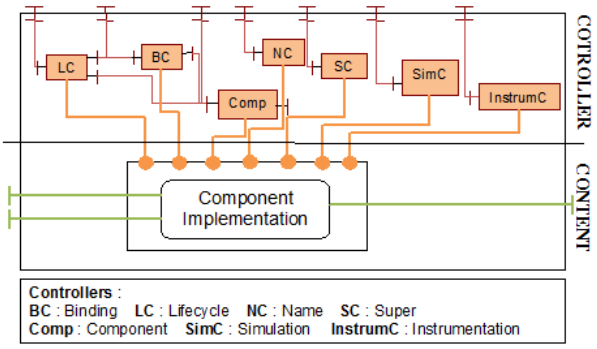

Fig. 5 Anatomy of an OSA component.

- the last form of distribution, which is somehow complementary of the previous ones, is achieved when the middle-ware is used to bridge together several simulation architectures, using the HLA standard for example[15].

Architecture Description Layer

So far, the standard Fractal ADL has been extended for the needs of OSA as follows:

- Event scheduling: in OSA an event corresponds to a method invocation (functor) scheduled at a given time in simulation. The event is scheduled directly at the level of the component that provides the method that will be called;
- Probes definition: as will be further explained in section 4, probes are used to collect the data samples needed for generating the outputs of the simulation. The ADL was extended to allow the specification in each component of the variables to observe in order to produce these samples.

Discrete-Event Engine Layer

The simulation engine is distributed over all component that have a surrounding membrane implementing the simulation non-functional services (represented by the "SimC" box on fig. 5). The simulation engine offers a process-oriented model of execution: a process corresponds to a living entity of the simulation (a thread). Each process has its own control flow, and may be interrupted on various blocking conditions (ie. on locks, waits and sleeps instructions). These blocking services are accessed through a dedicated `simulation-controller` interface (fig. 6). Component with a `simulation-controller` interface are called *active components* and those without such an interface are called *passive components*.

During the simulation, the functional part of the component (the model) may use the following services, that are provided by the `simulation-controller` interface:

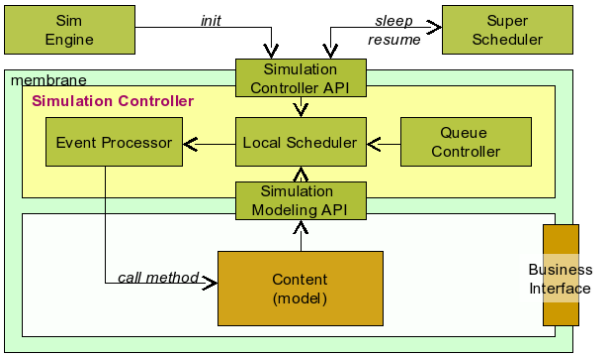

Fig. 6 Internal architecture of the simulation-controller.

- `current_time()`: returns the current simulated time;
- `abort()/terminate()`: requests abnormal/normal termination of the simulation execution;
- `object = wait(key, timeout)`, `release(key, object)`: `wait` blocks the current executing thread on a key object until another thread calls `release` with the same key object. Furthermore this wait/release mechanism allows the releaser to transmit an object reference to the waiter, a mechanism inspired from Hoare's Communicating Sequential Processes[16]; optionally, the waiting may be guarded with a timeout that sets the maximal simulated time after which the waiting thread must be woken up;
- `spin_lock()`, `spin_unlock()`, `spin_trylock()`: a basic locking mechanism for ensuring mutual exclusion inspired from the Linux kernel API[17]
- `schedule_myself(time, method, param)`: this primitive is used to schedule new events; indeed, in OSA, an event corresponds to the execution of a method at a specified time of the simulation.

However, this simulation API may be replaced or masked by another one. This is a powerful mean for reusing existing models developed for other discrete-event simulators that have their own different simulation APIs. In other words, OSA can easily *mimic* other simulators and therefore reuse their existing models. Moreover, the components used in a given simulation scenario are not forced to all share the same API which mean that, theoretically, components developed for various simulators may inter-operate in the same simulation scenario.

A second reason for ensuring such a versatile architecture is to allow for experimentations at the simulation engine level. In this case, OSA may be used as a testbed, for example to compare the performances of various distributed implementations of the engine.

4 The OSA Instrumentation Framework

The OSA Instrumentation Framework (OIF) is a new layer of the OSA architecture. The OIF is connected to the existing component hierarchy with a two-level structure: a lower *sampling* level and a higher *sample-processing* level, as shown on fig. 7. The sampling level is in charge of collecting data samples in the model components during the simulation runs. Data samples correspond to successive values taken by the observed objects during a simulation run. Each sample is time-stamped with the current simulated time or a sequence number.

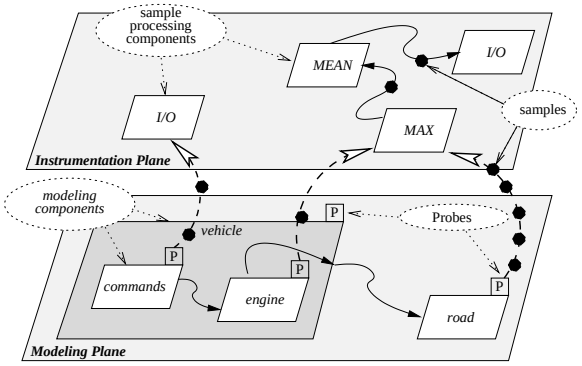

Fig. 7 OSA's Instrumentation Framework

At the lowest level, the data samples are automatically generated every time an observed object is modified, thanks to AOP techniques. The AOP technique currently used in OSA is based on the the AspectJ weaver and the AspectJ Java language extension[18]; it is further described hereafter in section 4.1.

The samples are directed to a dedicated controller, called the *probe*, which is placed in the surrounding membrane of each model component that owns at least one observed object. Thus, modeling components that are not instrumented don't embed any instruction related to instrumentation.

The probes in turn forward their samples to a central dispatching component called the *SuperSampler* (not represented on figure 7). This SuperSampler is a mediator between the two levels of the OIF. It is in charge of relaying the samples received from the probes to the processing components of the upper layer.

The components of the upper layer are in charge of the on-line statistical processing and implements Input/Output policies of samples. For this purpose we plan to reuse the COSMOS Fractal component framework[19].

The specification of which variables to observe is taken from an Instrumentation descriptor file. This file is an XML file that obey the OSA ADL specification, which is an extension of the Fractal ADL. This descriptor and the building process from the source files to the final simulation execution is then described in section 4.2.

4.1 Data samples generation

Without entering the details (see [18] for more details), the most important additional constructions offered by the AspectJ language are the following:

- Aspect: this construction resembles a Java class definition, with attributes and methods declaration, but with additional constructions, such as the ITD and Advice briefly described hereafter;
- Inter-Type Declaration (ITD): this construction is used to extend an existing type declaration (a Java class or Interface) with new elements (new attributes or methods);
- Advice: an advice is a piece of code that executes every-time a particular instruction of the initial program is reached. This particular instruction is specified using an AspectJ pattern-matching construction called a *pointcut*. An advice may be executed prior, in place of, or after the instruction identified by the pointcut.

In OSA, we use the AspectJ's advice construction to trap the successive values of the observed variables. For this purpose we need to define pointcuts that, for each observed variable, specify the time at which the observed variable is set. For example, in AspectJ, to specify that we want to trap the times at which the variable `X` of class `MobileObject` in package `example.package` is set, we use the following declaration:

```
public pointcut trap_X(float X):
    set(float example.package.MobileObject.X)
    && args(X);
```

Then, this pointcut may be used to define an advice, such as the following that calls the method `forward_to_probe(float val, int id)` which is supposed to forward the trapped value to the probe indicating the origin of the data with its `id`:

```
after(float X):trap_X(X) {
    forward_to_probe(X,MobileObject_X_ID);
}
```

In the previous example, notice the use of the `after` AspectJ keyword, that means that the corresponding code must be executed after the value of the observed variable `X` is changed.

4.2 OSA Building Workflow with Instrumentation

The OSA building workflow with Instrumentation is depicted on figure 8.

The building workflow starts with the following set of source files and XML descriptors, that are either produced manually or through a Eclipse-based helper or editor[1] :

- The Instrumentation Descriptors are XML files that obey the OSA ADL syntax. They list the observed objects in each OSA model component. Any Java language type may be observed. However, non primitive types need to be extended (using an AspectJ ITD) with a sampling function that computes the value of the sample. This function is provided in a separate source file or library;
- The Models sources and Engine Sources;
- The Deployment Descriptors are another type of XML files that describe the location of the physical computational resources requested for running the simulation. This file is not an ADL file because it is never read by the OSA ADL factory; it is read prior to the start of the execution of the OSA simulation, and thus before the OSA ADL factory is created. This file is used by the OSA deployment facility to prepare the distribution and execution of the OSA components (including the ADL factory) in several (distributed) Java Virtual Machines.
- The Model Descriptors are XML files that obey the OSA ADL syntax. They describe the topology of the components that describe the Model of the system that is to be simulated. The result is a hierarchical component set whose root represents the whole Model of the System to be simulated. The previously described Instrumentation Descriptors are tightly connected to this Model Descriptors in the following manner: the root of Model is attached to the root of the Instrumentation in such a way that the result is a new Fractal application that contains both the Model components and the Instrumentation components. Hence, the two level of the architecture depicted on figure 7.

The Listing 6 shows an example of an Instrumentation ADL descriptor. This example instruments a variant of the Fractal "HelloWorld" example presented in section 4.1. The modification is such that the component named "Client" has an additional `X` attribute of type `float`. The first part of the listing shows that the root definition of this ADL extends the "HelloWorld" ADL presented in Listing 5. Then, the definition of the "Client" component is extended with a new `probe` that contains one `statevar` declaration. Indeed, recall that, in each component, a probe may collect the samples generated from several observed objects. In our case, the only observed object is the `ClientImpl.X` attribute of type `float`.

Then, the second part of this listing shows that a new component, named `sampleMean`, is added in hierarchy defined by the ADL. This new component is a sample-processing component. It lays in the upper-level of the OIF. This sample-processing component contains a `sampleprocessor` definition that references the state-variable declared in the previous probe. This is how the connection between the probe and the sample-processing component is established.

[1] At the time of writing, Eclipse support is not yet available for all these files

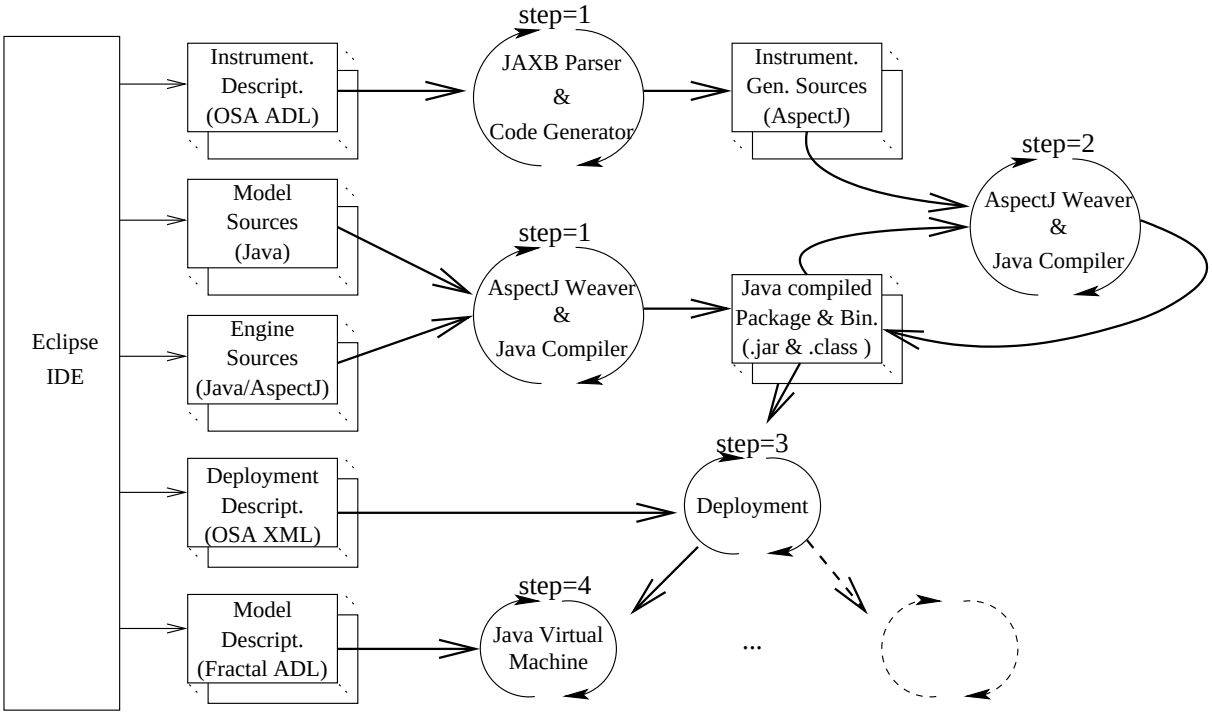

Fig. 8 OSA Building Workflow with Instrumentation

The first step of the simulation building workflow consists in compiling the Java sources and weaving the AspectJ source files of the Models and simulation engine, and to produce the AspectJ code of the probes. This latter code is automatically generated by a custom generator that parses the Instrumentation Descriptors with an XML parser based on the JAXB library. Then, a second step of compilation and weaving is required, in order to compile the previous generated code. The third step consists in parsing the Deployment Descriptor in order to prepare for the (possibly) distributed execution of the components. This distributed execution is handled by the FractalRMI contributed middle-ware library[2]. Finally, the Fractal ADL factory component is started and starts reading the Instrumentation ADL which, in turn, references the Modeling ADL. This workflow is handled thanks to the Maven building system, that automatically solves any dependency issue.

5 Conclusion

In this paper we presented our ongoing work on building an Instrumentation Framework by applying the "Separation of Concerns" paradigm. The result of this design is that the OSA model components can be instrumented without any modification to their code. This Separation of the Modeling and Instrumentation concerns has many benefits.

First, when a given model is reused several times in the same simulation scenario, this model may be instrumented independently and differently each time it is instantiated in the scenario. Second, since the instrumentation is fully stored apart from the model, several instrumentations can be build concurrently without interfering with each other. This is useful for example for studies that require many simulations with various instrumentations. Third, the AOP technique described in this paper should theoretically lead to a minimal instrumentation overhead without introducing a limitation on the number and types of objects that can be instrumented. However, this last claim about performance still needs to be demonstrated. Last, it should be noted that despite the OIF is part of the OSA Architecture, it is rather loosely coupled to the rest of the architecture and should therefore be easily reusable in most Fractal applications. Indeed, instrumenting a simulation application or a general purpose application is very similar.

Our future development directions will be to implement an Eclipse-based user friendly interface for the OIF. Indeed, the OIF is currently operational[3] but in order to build new instrumentations the end-user has to write the ADL/XML descriptors files manually.

Our future research directions will be to work on the deployment and performance optimization issues. Indeed, the data flows generated by the instrumentation during a simulation are very important and rapidly increase as the number of simulated entities increases. Since we plan to run distributed simulations of very large sys-

[2]Available on the Fractal's forge at `http://fractal.objectweb.org/`.

[3]The OSA software is available under LGPL license from the INRIA forge at `http://osa.gforge.inria.fr`. See also the OSA wiki web site for general information about the project: `http://osa.inria.fr/`.

Listing 6 An example of an Instrumentation Descriptor file

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
          "classpath://osa/util/adl/stdsim.dtd">


<definition name="cs.adl.Instrumentation" extends="HelloWorld">
    <component name="Client" definition="ClientImpl">
        <probe>
            <statevar name="ClientImpl.X" type="float"/>
        </probe>
    </component>

    <component name="sampleMean">
        <interface name="sampleprocessorItf" role="server" signature="<...>.SampleProcessorItf" />
        <content class="osa.util.operations.SampleProcessorMean" />
        <controller desc="primitiveSim" />
        <sampleprocessor signature="sampleprocessorItf">
            <statevar name="ClientImpl.X" componentname="client" type="float"/>
        </sampleprocessor>
    </component>

</definition>
```

tems such as Peer-to-Peer networks systems, this issue of performance and scalability will become our major concern.

Acknowledgments

This work is partly co-supported by the IST-FET "AEOLUS" project, the ANR "OSERA" grant and INRIA.

References

[1] Mehmet Aksit. Separation and composition of concerns in the object-oriented model. *ACM Computing Surveys*, 28(4es):148, December 1996.

[2] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154, 1996.

[3] Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[4] George S. Fishman. *Discrete-Event Simulation – Modeling, Programming, and Analysis*. Springer, 2001.

[5] Olivier Dalle. OSA: an Open Component-based Architecture for Discrete-event Simulation. Technical Report RR-5762, INRIA, February 2006. Available from http://www.inria.fr/rrrt/rr-5762.html.

[6] Olivier Dalle. The OSA Project: an Example of Component-Based Software Engineering Techniques Applied to Simulation. In *Proc. of the Summer Computer Simulation Conference (SCSC'07), San Diego, CA*, July 15-18 2007. Invited paper. To appear.

[7] Open Simulation Architecture (OSA), a collaborative platform for component-based discrete-event simulation. Web page at http://osa.inria.fr/.

[8] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 11-12(36), 2006.

[9] Lionel Seinturier, Nicolas Pessemier, Laurence Ducgien, and Thierry Coupaye. A component model engineered with components and aspects. In *Proc. of the 9th Int'l Symp. on Component-Based Software Engineering*, volume LNCS 4063, pages 139–153, Vasteras, Sweden, June 29th - July 1st 2006. Springer.

[10] Steve Holzner. *Eclipse*. O'Reilly, May 2004.

[11] Jim des Rivières and John Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

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

[13] Erich Gamma and Kent Beck. *Contributing to Eclipse: principles, patterns, and plugs-in*. The Eclipse series. Addison-Wesley, 2004.

[14] Richard M. Fujimoto. *Parallel and distributed simulation systems*. Wiley Series on Parallel and Distributed Computing. J Wiley & Sons, 2000.

[15] IEEE-SA. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA), Federate Interface Specification*. Std 1516.1-2000.

[16] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[17] D. (Daniele) Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly & Associates, Inc., 2000.

[18] Russel Miles. *AspectJ Cookbook*. O'Reilly Associates, 2004.

[19] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Scalable Processing of Context Information with COSMOS. In *Proc. of 7th IFIP Int'l Conf. on Distributed Applications and Interoperable Systems*, June 2007.

