OSIF: A Framework To Instrument, Validate, and Analyze Simulations

Judicaël Ribault - Olivier Dalle
INRIA - CRISAM
University of Nice Sophia Antipolis
I3S-UMR CNRS 6070
BP93 - 06903 Sophia Antipolis, France
judicael.ribault@sophia.inria.fr
olivier.dalle@sophia.inria.fr

Denis Conan - Sébastien Leriche
Institut Télécom, Télécom SudParis
UMR CNRS SAMOVAR
9 rue Charles Fourier
91011 Évry, France
denis.conan@it-sudparis.eu
sebastien.leriche@it-sudparis.eu

ABSTRACT

In most existing simulators, the outputs of a simulation run consist either in a simulation report generated at the end of the run and summarizing the statistics of interest, or in a (set of) trace file(s) containing raw data samples produced and saved regularly during the run, for later post-processing. In this paper, we address issues related to the management of these data and their on-line processing, such as: (i) the instrumentation code is mixed in the modeling code; (ii) the amount of data to be stored may be enormous, and often, a significant part of these data are useless while their collect may consume a significant amount of the computing resources; and (iii) it is difficult to have confidence in the treatment applied to the data and then make comparisons between studies since each user (model developer) builds its own ad-hoc instrumentation and data processing. In this paper, we propose OSIF, a new component-based instrumentation framework designed to solve the above mentioned issues. OSIF is based on several mature software engineering techniques and frameworks, such as COSMOS, Fractal and its ADL, and AOP.

Categories and Subject Descriptors

I.6.6 [**Simulation And Modeling**]: Simulation Output Analysis; I.6.4 [**Simulation And Modeling**]: Model Validation And Analysis; D.2.13 [**Software**]: Reusable Software

General Terms

Design, Experimentation, Measurement, Verification

Keywords

Instrumentation, Observation, Context management, Aspect Oriented Programming

1. INTRODUCTION

SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.

The workflow used for studying a system using discrete-event simulation is often described in the simulation literature, e.g. in [2, 12]. Despite a few minor differences, every author seems to agree on the various major steps of this workflow: define the objective(s) of the study, collect data about the system to be simulated, build a model of the system, implement an executable version of this model, verify correctness of the implementation, execute test runs to validate the simulation model, build experiment plans, run production runs to generate outputs, analyze data outputs, and finally, produce reports.

In [18], Zeigler et al. further refine the methodology by introducing the concept of Experimental Frame as follows: "[An experimental frame] is a specification of the conditions under which the system is observed or experimented with". Hence, their Experimental Frame not only describes the instrumentation and output analysis but also drives the simulation. Thanks to this separation between the Experimental Frame and the system model, it is possible to define many Experimental Frames for the same system or apply the same Experimental Frame to many systems. Therefore, we can have different objectives while modeling the same system, or have the same objective while modeling different systems.

In [7], Dalle and Mrabet already presented the OSA Instrumentation Framework (OIF). OSA [6] stands for Open Simulation Architecture. OIF is inspired from the concepts of the DEVS Experimental Frame (EF) but it only focuses on the instrumentation, validation and analysis concerns. Indeed, in OSA, the instrumentation and scenario concerns are separated into distinct layers which is not the case in the DEVS EF. On the contrary, the DEVS EF specifies three distinct entities (generator, transducer and acceptor), and therefore establishes a clear separation between three concerns, that are not distinguished in OSA. However, the concept of layers found in OSA is orthogonal to that of entities (or component, which are also supported by OSA), which means that OSA could actually implement both separations —i.e., in OSA, one can easily implement the generator, acceptor and transducer components in *both* the scenario and instrumentation layers.

In this paper we present the Open Simulation Instrumentation Framework (OSIF). OSIF is inspired from the OIF project but it is not a part of the OSA project. In fact, one of our motivation is the ability to plug OSIF on any simulator. We use our experience in building the OSA architecture to

build a new framework dedicated to the instrumentation of simulations and to the processing of simulation data, based on similar concepts: provide an open architecture, with clear separation of concerns, and in the end, favor reuse of useful components. OIF is a tool for OSA while OSIF aims at being a generic instrumentation and data processing framework that could be plugged onto any simulation code (including OSA) written in a language supporting AOP (Java, C/C++, Python...). The simulation code does not need to be available in source form since AOP can also be applied on compiled code or at runtime.

Indeed, like in OIF, in OSIF we propose to separate the modeling concern and the instrumentation concern using AOP (bottom-half of the framework), and to do on-line statistics computation and to transmit the data across the network to their storage destination using COSMOS (top-half of the framework). Moreover, it is possible to save the simulation data in any format. For example, using the standard format of the OMNeT++ simulator, we are able to reuse the Scave tool to post-process simulation data.

In the sequel of the paper, Section 2 presents our motivations to build a generic open instrumentation framework and how we plan to achieve our goals. Section 3 introduces the case study used throughout the paper. Then, Section 4 presents the generic instrumentation framework illustrated through a large-scale simulation case study. Section 5 compares the contribution of this paper to related works. Finally, Section 6 concludes the paper and draws some perspectives.

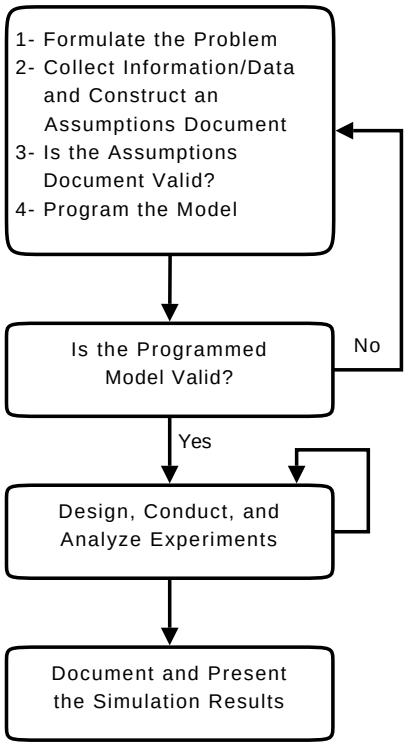

Figure 1: Simulation workflow focusing on instrumentations tasks.

2. MOTIVATIONS AND OBJECTIVES

Law presents in [12] a general workflow to build a valid and credible simulation study. Figure 1 resumes Law's workflow, focusing only on tasks where instrumentation is needed. The first four tasks of the Law's workflow lead to a simulation model. Then, the programmed model is validated: the simulation model is instrumented and validation results are compared with results from the real system. Next, more simulation experiments are designed, conducted and analyzed accoording to the simulation objectives. The conclusions drawn from the simulation results are finally presented in a document.

In this paper the term simulation model is used to designate the source code that implements the behavior of the model of the system under study and the term system model is used to designate the conceptual model of the system under study. In the following sections we further discuss the four main reasons that motivate our work: the ability to separate concerns (sec. 2.1), the ability to apply the same validation process to real and simulated systems (sec. 2.2), the ability to easily switch from live to post-mortem analysis (sec. 2.3) and the need for composition in data processing (sec. 2.4).

2.1 Separation of Concerns

To design the instrumentation of a simulation model by following the workflow depicted in Fig. 1, most simulators offer a simulation API for the declaration of observable data within the simulation model. These common practices imply that all the possible observations for a given simulation model need to be decided (and hard coded) at the time the simulation model is implemented. Hence, simulation model developers may find it difficult to choose which data need to be instrumented while experimenters may be reluctant to run simulations that generate large amount of useless data. Moreover, if the resulting simulation model does not contain the required instrumentation for an analysis, a software evolution is necessary to modify the simulation model. This raises an issue about the credibility of the conclusions drawn from the comparison of simulation results obtained using different simulation models.

From this perspective, the separation of concerns between model and instrumentations provides many benefits. For example, keeping the simulation model clear from any instrumentation allows to reuse it in every kind of studies and makes it more understandable. Moreover, instrumenting only the data needed allows to run simulations faster. On large-scale simulations involving many experts, each expert could work on his/her part. Indeed, it is important for large-scale and distributed simulations to allow instrumentation experts working independently from the simulation modeling expert. This leads to remove dependences between tasks of Box 1 and the other boxes in Figure 1.

We propose to use the Aspect-Oriented Programming (AOP) paradigm [11] because it allows such a separation of the modeling and instrumentation concerns, each of which being weaved on demand, possibly at the last minute, before an actual simulation run execution is started. Moreover, we separate the collection of raw instrumentation data (into collectors) from the processing of higher-level instrumentation results (into processors with generic operators). This second separation of concerns is one of the key concepts proposed by the COSMOS framework [5, 16].

2.2 From Real to Virtual System

Before carrying out a simulation study, it is necessary to follow a validation process as mentioned in the second box of Figure 1. The validation process can help validate a simulation model by comparing simulation results with experiments on a real system. This requires to use firstly the same inputs and secondly the same statistical analysis on the outputs. The best approach would be to use the same processing workflow on both the simulation and real data. Indeed, a process to validate results that could be applied both on a real system and on a simulation model gives more credibility to the simulation model, and allows extrapolating the findings of a simulation on a real system.

The COSMOS framework has been created to process context information of real systems during their executions. Using also COSMOS as the data processing framework for simulation purposes allows using it both when experimenting the real system and simulating the virtual system.

2.3 From Live to Post-mortem analysis

The third box of Figure 1 is about the design, the running and the analysis of simulation experiments. Running a simulation may result in a huge amount of simulation data and may then consume a lot of disk space. Moreover, gathering data in a distributed simulation is not trivial and may also consume a lot of network bandwidth. After having run experiments, when all the simulation data are collected, a validation phase is necessary before they are analyzed. Indeed, a simulation run may produce results that could not be analyzed for instance because the confidence interval is too large or because the duration of the simulation considering the simulation time is too narrow. If so, it is necessary to loop to the third box in order to obtain results that are analyzable. Afterwards, simulation results are analyzed and conclusions can be drawn (fourth box of figure 1).

In order to avoid memory, disk, and bandwidth consumption issues, and in order to ease and then optimize the validation and analysis processes, we propose to execute these three steps (data gathering, validation of simulation results, and analysis) together during the simulation run (called live analysis). Therefore, data gathering may not need to store data on disks but send them instead directly to the validation and analysis processors. The validation process dynamically controls the analysis process in order to produce results easily understandable. As a consequence, the data flow can be optimized. Moreover, it is easier to replay a study that integrates its data processing. Indeed, since statistical analysis are done in live during the simulation run, no third-party tool is required. Nevertheless, it is sometimes necessary to preserve raw data (e.g. for debugging purposes). Thus, logging capabilities for post analysis is also a requirement.

COSMOS provides the developer with pre-defined generic operators such as averagers or additioners. Each operator is included into a unit of control called a processor or a node. A node can be finely tuned to be active or passive, blocking or not in observation or in notification, etc. Therefore, COSMOS allows us to easily build a live analysis on instrumented data while optimizing data flow. Moreover, we will show in the next sections how we can easily complement chains of processors that optimize data flows for live analysis with chains of nodes that log raw data for post analysis.

2.4 Data Processing Composition

Another interesting feature is the possibility to write simple instrumentation and data processing (validation of simulation results and analysis) and combine them into more complex ones. Benefits are valuable since writing many simple instrumentation and data processing is easier than writing a complex one. Considering reuse, it is more likely to reuse several times simple and generic data processing rather than reusing a complex dedicated one. Another case of reuse is the design of a new data processing from an already existing one from a catalogue. Reusing and composing data processing is also an asset when comparing studies sharing the same data processing. In that case, it is easier to compare and trust the results because validation of simulation results and results analysis are the same among studies. Of course, since instrumentation makes the bridge between simulation model and data processing, instrumentation reuse is possible as long as the code onto which instrumentation is applied is similar with the one for which the instrumentation was initially designed for. On the other hand, the data processing is independent and can be reused regardless the language and the simulation model. Thus, at worst it will be necessary to create/adapt the instrumentation code, but we can always reuse the data processing.

3. CASE STUDY

As a proof of concept, we have decided to use OSIF to instrument and to process data on a complex simulation. We choose the simulation of a peer-to-peer system[1] because it shows the limits of simple instrumentation frameworks.

The goal of the simulation case study is to simulate a safe backup storage system on a peer-to-peer network. The model involves N peers and one super-peer connected through a network. The super-peer has a global vision of the system like in the Edonkey2000 protocol. Each peer could establish communication with any other peers ($N * (N-1)$ links). The scenario involves users. Each user is connected to one peer and can push data into the P2P storage system. Data are split into blocks of the same size, each block is fragmented into s fragments. From these s fragments, r redundancy fragments are computed. The $(s+r)$ fragments of the data block are distributed on different peers. Any combination of s fragments allows to rebuild the raw data. Therefore, the system tolerates r failures. Peers are free to leave the system at any time. Peers who disappear are considered dead and reappear empty after a certain period. In that case, a reconstruction mechanism of lost fragments is introduced to ensure data durability.

From this model, several instrumentations can be conducted. For instance, to validate results of the simulation model, we want to check that the peer's lifetime corresponds to the values that have been specified, to check that the number of fragments corresponds to the number of blocks into the system multiplied by $(s+r)$, or to check that the reconstructions only processes critical blocks. When the simulation model is validated, many studies can be conducted, for instance for analyzing the incidence of peers' lifetime, redundancy level, topology of the network of peers, or blocks' allocation on peers.

For the sake of simplicity, we will focus only on the instrumentation of the peers' lifetime that illustrates issues

[1] `http://www-sop.inria.fr/mascotte/Contrats/spreads`

about separation of concerns, optimization of instrumented data flow, and design of complex data processing by composition. To prototype our solution, we have used the OSA simulator.

4. OPEN SIMULATION INSTRUMENTATION FRAMEWORK

In this section, we present the Open Simulation Instrumentation Framework (OSIF). We begin with a short introduction to the principles of the COSMOS framework. Then, we explain how to handle issues presented in Section 2 with OSIF.

4.1 COSMOS

COSMOS (COntext entitieS coMpositiOn and Sharing) [5, 16, 15] is a LGPL component-based framework for managing context data in ubiquitous applications; it is based on the FRACTAL component model [3, 4]. Context management is (i) user and application centered to provide information that can be easily processed, (ii) built from composed instead of programmed entities, and (iii) efficient by minimizing the execution overhead. The originality of COSMOS is to use a component-based approach for encapsulating fine-grain context data, and to use an architecture description language (ADL) for composing these context data components. By this way, we foster the design, the composition, the adaptation and the reuse of context management policies. In the context of OSIF, context data components become instrumentation data components.

The COSMOS framework is architectured around the following three principles that are brought into play in OSIF: the separation of data gathering from data processing, the systematic use of software components, and the use of software patterns for composing these components. The first principle supports the clean separation between data gathering that may depend upon the simulation framework and data processing that is simulation framework agnostic. The second principle, software components, fosters reuse everywhere. Finally, the third principle promotes the architecture-based approach "composing rather than programming". The COSMOS framework is implemented on top of the FRACTAL component ecosystem with the message oriented middleware (MOM) DREAM [14].

4.2 Separation of Concerns

In OSIF, the separation between simulation and instrumentation concerns is performed using the Aspect-Oriented Programming (AOP) paradigm and the COSMOS' concept of the "collector".

Aspect-Oriented Programming.

AOP [11] is a software engineering technique for modularizing applications bringing many concerns into play. The general idea is that, whatever the domain, large applications have to address different concerns such as data management, security, GUI, data integrity. Using only procedural or object orientations, these different concerns cannot always be cleanly separated from each other and, when applications evolve and become more complex, concerns end up being intertwined, called the "spaghetti" code problem. AOP promotes three principles. Firstly, functional or extra-functional aspects of an application should be designed independently from an application core and so the application design is easier to understand. Secondly, it is not easy to modularize common-interest concerns used by several modules, like logging service. Those cross-cutting concerns can be described using AOP cross-cutting expressions that encapsulate each concern in one place. Thirdly, AOP favors inversion of control principle. Inversion of Control (IoC) is a design pattern attempting to remove all dependencies from the business code by putting them in a special place where the goal is only to manage dependencies. Considering a simple example of a lamp controlled by a switch, in basic object-oriented programming, the control of the lamp is placed in the code of the switch. Using the inversion of control principle with AOP the control of the lamp is no longer in the code of the switch but in a dedicated aspect that will make the connection between the switch and lamp. This results in a better separation of concerns and reusability.

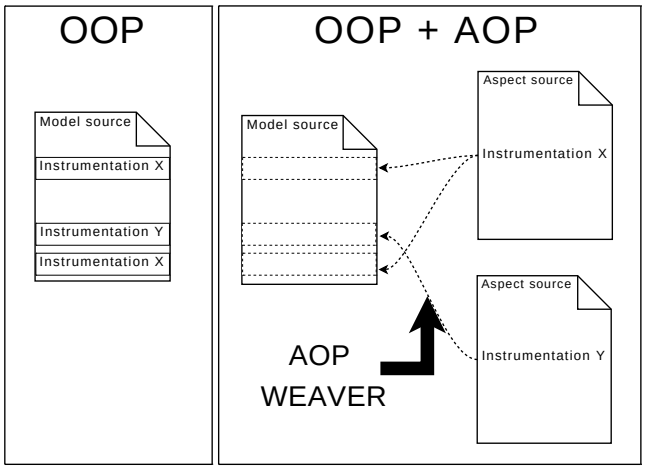

Figure 2: Separation of concerns using AOP.

Instrumentation is a cross-cutting concern because many parts of the simulation model need to be introspected and complemented with instrumentation concern. The left part of Figure 2 illustrates how the instrumentation concern pollutes the modeling code when using traditional Object-Oriented Programming (OOP). The code is hard to read and it is hard to figure out where the code of the model is. The right part of Figure 2 illustrates how to avoid this drawback by reorganizing the instrumentation concern into separate source code files (aspects). The arrow represents the action of the AOP weaver which is a tool that is responsible for binding the aspects with the modeling code on demand (and possibly dynamically). This keeps the code of the model concise and stripped from the instrumentation code.

As an example, Listing 1 illustrates a Java class Peer with two methods: boot() and halt(). Each method has some modeling code and call the instrumentation framework (crosscuting concern). The instrumentation code pollutes the modeling code as illustrated on the left part of Figure 2. Indeed, Java class Peer invoke method write(message) which is part of the instrumentation framework represented here by the Sampler object. Sampler is connected to the simulator engine and writes on disk the simulation time and the message.

Listing 1: Peer Java class without separation of concerns.

```
public class Peer{
  Sampler sampler;
  String peername;

  public void boot()
  {
    [...] //modeling code snipped
    sampler.write(peername+" boot");
  }

  public void halt()
  {
    [...] //modeling code snipped
    sampler.write(peername+" halt");
  }
}
```

Thanks to AOP, it is possible to separate modeling concern from instrumentation concern. The aspect of Listing 2 written in AspectJ shows how to isolate the instrumentation concern in a separate module as illustrated on the right part of Figure 2. The aspect peer_instrumentation calls the instrumentation framework right after the execution of methods boot() and halt().

Therefore, Listing 3 illustrates the same modeling code stripped from instrumentation concern.

Listing 2: AspectJ aspect to observe Peer class.

```
public aspect peer_instrumentation {
  Sampler Peer.sampler;

  after(Peer peer): execution(void Peer.boot())
                    && this(peer)
  {
    sampler.write(peername+" boot");
  }

  after(Peer peer): execution(void Peer.halt())
                    && this(peer)
  {
    sampler.write(peername+" halt");
  }
}
```

Since only the required instrumentation aspect is weaved to the simulation model, the execution of the simulation runs faster. Moreover, software evolutions of the simulation model and the instrumentation are facilitated. Finally, data processing can be developed independently by instrumentation experts and reused more easily.

Listing 3: Java class with separation of concerns.

```
public class Peer{
  String name;

  public void boot()
  {
    [...] //modeling code snipped
  }

  public void halt()
  {
    [...] //modeling code snipped
  }
}
```

COSMOS Collector.

The lower layer of the COSMOS framework defines the notion of a data collector. In the context of ubiquitous applications, data collectors are software entities that provide raw data from the environment. In M&S, the environment is the simulated model under study. A data collector retrieves instrumented data from a simulation and provides them to the data processors. COSMOS collectors are generic and the data structure to be pushed by the advice code of the instrumentation aspect is an array of Object. Therefore, COSMOS collectors and AOP instrumentation advices perform the junction between the simulation framework and instrumented data processors of OSIF.

4.3 From Live to Post Analysis

We propose to reuse some concepts of COSMOS such as the data processor and the data policy to analyze simulation data in live but also to log the raw simulation data while preserving optimization on data flow.

COSMOS processor.

We have seen that the lower layer of the COSMOS framework defines the notion of data collector. The middle layer of the COSMOS framework defines the notion of a data processor, named context processor in COSMOS. Data processors filter and aggregate raw data coming from data collectors. The role of a data processor is to compute some high-level numerical or discrete data from raw numerical data outputted either by data collectors or other data processors. Therefore, data processors are organized into hierarchies with the possibility of sharing. A data processor or node of this graph can be parameterized to be passive or active, blocking or not in observation or in notification.

- **Passive Vs. active.** A passive node obtains simulation data upon demand; a passive node must be invoked explicitly by another node. An active node is associated to a thread and initiate the gathering and/or the treatment of simulation data. The thread may be dedicated to the node or be retrieved from a pool. A typical example of an active node is the centralization of several types of simulation data, the periodic computation of a higher-level simulation data, and the provision of the latter information to upper nodes, then isolating a part of a hierarchy from too frequent and numerous calls.

- **Observation Vs. notification.** The simulation reports containing simulation data are encapsulated into messages that circulate from the leave nodes to the root nodes of the hierarchies. When the circulation is initiated at the request of parent nodes or client applications, it is an observation. In the other case, this is a notification.

- **Blocking or not.** During an observation or a notification, a node that treats the request can be blocking or not. During an observation, a non-blocking node begins by requesting a new observation report from each of its child nodes, and then updates its simulation data before replying to the request of the parent node or the client application. During a notification, a non-blocking node computes a new observation report with the new simulation data just being notified, and then notifies the parent node or the client application. In the case of a blocking node, an observed node provides the most up-to-date simulation data that it

possesses without requesting child nodes, and a notified node updates its state without notifying parent nodes.

Figure 3 illustrates how to use COSMOS data processors to collect and compute outputs in a distributed simulation. A distributed simulation involving three peers is executed on two computers. This processing produces as output the min, max, and average lifetime of peers. Each peer is connected to a data collector. Data collectors receive simulation data every time the state of the attached peer changes. Then, collectors push the simulation data to the data processors in which they are enclosed. Data processors O3, O5, and O6 compute the lifetime of peers, that is the difference between the starting up and the shutting down, and send these simulation data to processors O2 and O4. Those processors gather these simulation data from all the peers of the same host and compute the min, max, sum of the lifetimes of peers and the total number of times lifetime has been calculated. Since the latter processors are blocking in notification, the flow of data is stopped. In conclusion, data processors O2 or O4 are updated every time a peer lifetime is computed and gather simulation data collected on every peer executing in a host.

Data processor O1 is responsible for gathering simulation data at the global level, that is for all the peers of all the hosts. This node is active in observation but blocking, thus meaning that it periodically requests simulation data from data processors 02 and 04.

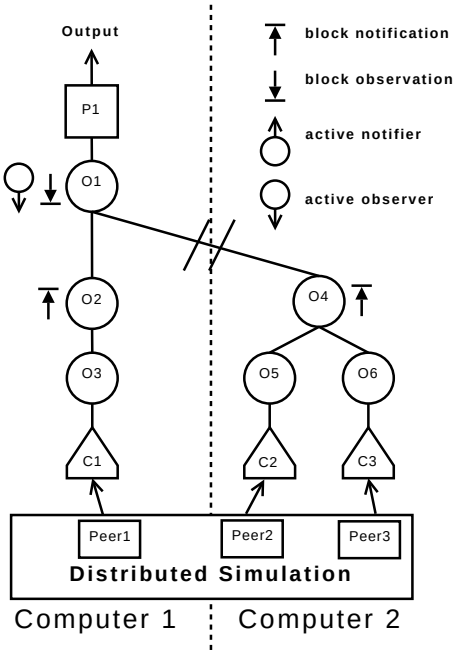

Figure 3: Graphical representations of data processing in a distributed simulation.

Therefore, considering this live analysis, the disk overhead to store simulation data can be minimum if we store only the final result provided by node O1. Concerning the bandwidth overhead, it depends upon the number of requests performed by node O1 and upon the amount of simulation data transferred from node O4 to O1. Considering the previous simulation involving N peers (N' is the number of peers on Computer2), each peer being started up and shut down T times during the simulation. A basic instrumentation would have written $N * T$ times the peer name, the action (boot or halt) and the simulation time on disk. The same instrumentation would have transferred $N' * T$ times the peer name, the action (boot or halt) and the simulation time through the network. For large-scale simulations, the amounts of data will be huge. Using OSIF, we can easily build a live analysis instrumentation that directly writes only the min, max, and average lifetimes of peers on disk, and transfers the min, max, and sum of these lifetimes and the total number samples used for the computation through the network. Moreover, live analysis has its own separate thread, running asynchronously, which proficiently benefits from the parallel computing capabilities of modern multicore architectures. For example, running a local simulation involving 1000 peers during 5 years takes 23 seconds to instantiate and 70 seconds to execute with live analysis of peers lifetime. With basic logging of the raw simulation data the same simulation takes 14 seconds to instantiate and 70 seconds to execute. The difference represents the overload due to the instantiation of COSMOS nodes and collectors.

COSMOS instrumentation policy.

The upper layer of the COSMOS framework defines the notion of a context policy that translated into the concept of instrumentation policy in OSIF. COSMOS instrumentation policy abstracts simulation data provided to the user/application, that is instrumentation policies are the "entry points" to the graph of processing nodes. We use instrumentation policies to translate instrumentation data provided by COSMOS processors into an understandable format: textual, graphical, or third-party tools compliant.

Let's take the example of Figure 3. The goal now is to log simulation data in order to have the complete peers' connection and disconnection history. The node O1 aggregates and merges simulation data from nodes O2 and O4 and pushes them to the instrumentation policy P1. The instrumentation policy may then decide the output format, for instance for being able to process them using OMNeT++ analysis tool Scave [17]. Scave helps the user to process and visualize simulation results saved into vector and scalar files. So, the instrumentation policy P1 translates simulation data outputted by node O1 into vector or scalar files understandable by Scave. Thus, we can later post-analyze our data using Scave.

We have seen how using COSMOS data processors and policies can help in designing a live analysis or a logging system. Logging raw data is necessary in certain cases such as debugging but live analysis allows for disk usage and network bandwidth usage optimizations. The CPU overhead may be significant on large instrumentations with lots of computations: in the worst case, it will be the same as the computation needed by a post analyze. Thus, OSIF allows designing instrumentations taking into account the topology of the simulation and optimizing the data flow, but it can also produce results consistent with existing tools in order to compare simulation studies' results.

4.4 Composition of instrumentations

In this section, we show how to use the architecture de-

scription language of FRACTAL (FRACTAL ADL) for sharing, reusing and mixing COSMOS-based instrumentation processing: collectors, data processors, and instrumentation policies.

Component-based Architecture.

Being based on COSMOS, OSIF benefits from the three principles of separation of concerns, isolation and composability of the component-based software engineering approach: in COSMOS, every data collector and every data processor is a software component. By connecting these components, we define assemblies that gather all the information needed to implement a specific instrumentation policy. COSMOS is implemented with the FRACTAL component model and instrumentation policies are specified using FRACTAL ADL. Designers of intrumentation policies are able to describe complex hierarchies of data processors by taking advantage of the two main characteristics of FRACTAL: hierarchical component model with sharing.

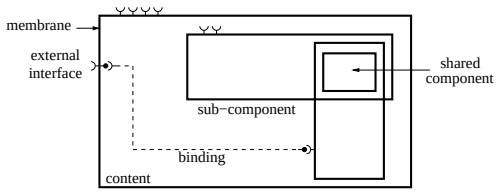

Figure 4: Graphical notations of the Fractal component model.

As depicted in Figure 4, a component is a software entity that provides and requires services. The contracts of these services are specified by interfaces. FRACTAL distinguishes server interfaces that provide services and client interfaces that specify the required services. A component encapsulates a content. It is then possible to define hierarchies of components offering views at different levels of granularity. Hierarchies are not solely trees since sub-components are shared by several composites. The notion of sharing of components naturally expresses the sharing of system resources (memory, threads, etc.). Components are assembled with bindings, which represent a communication path from a client interface to a (conformant) server interface. Moreover, compositions of components are described with an Architecture Description Language (ADL). Without lack of generality, in the sequel, we use the graphical notations of the FRACTAL component model, as illustrated in Figure 4, and the language FRACTAL ADL [13] that is based on XML.

Architecture Description Language.

FRACTAL ADL is a XML language to describe the architecture of a FRACTAL application: components' topology (or hierarchy), relationship between client interfaces and server interfaces, name and initial value of components' attributes. A FRACTAL ADL definition can be divided into several sub-definitions placed into several files. Thus, FRACTAL ADL allows the separation of concerns because application definition can be split into multiple files. Moreover, FRACTAL ADL supports a mechanism to ease the extension and redefinition through inheritance. Extension and redefinition allow the reuse (of a part or the whole) of existing instrumentation policies written using FRACTAL ADL. When a definition B extends a definition A, B possesses all the elements defined in definition A, like an internal copying mechanism. Moreover, if definition B defines an element that has the same name in definition A, B's definition overrides A's one. The extension mechanism enables us to create a new definition by composition of existing definitions. To manage Fractal components at a conceptual level, it is possible to use a graphical user interface integrated in Eclipse. Listing 4 illustrates a FRACTAL ADL definition of the live analysis of a peer. This definition takes one argument and is composed of four FRACTAL components: a collector and a data processor parameterized with the name of the peer used in argument and allowing to compute the lifetime of a peer, a data processor computing the average lifetime, and an instrumentation policy presenting the result.

Listing 4: Fractal ADL definition of a live analysis of a peer lifetime.

```
<definition name="PeerLifetime" arguments="
    peername">
  <component name="OutputPolicy"
    [...] <!-- ADL code snipped -->
    <component name="AverageLifetime"
      [...] <!-- ADL code snipped -->
      <component name="LifetimeOf${peername}"
        [...] <!-- ADL code snipped -->
        <component name="CollectorOf${peername}"
          [...] <!-- ADL code snipped -->
        </component>
      </component>
    </component>
  </component>
</definition>
```

Figure 5 illustrates the multiple extension capability of FRACTAL ADL. At the top, we have a FRACTAL ADL definition extending the FRACTAL ADL definition of Listing 4 with two different parameters. At the bottom, we have the resulting instrumentation policy. We can see that the data processors LifetimeOfPeer1 and LifetimeOfPeer2 are both encapsulating the data processor AverageLifetime. This can be done thanks to the inheritance mechanism of FRACTAL ADL. From this example, it's easy to imagine and design more complex compositions such as composition of several live analysis (lifetime, bandwidth...) and logging for later post-analysis of the simulation data.

The extension capabilities of FRACTAL ADL allow, as illustrated in the example defined in figure 5, to merge components. The merging process allows to update and extend already defined component. Therefore, it is possible to reuse and extend existing data processing and configure it. This can range from a simple update of parameters until the replacement or addition of management contexts.

There are several benefits to using the extension and the redefinition mechanisms of FRACTAL ADL. Firstly, writing many simple instrumentations and data processings is easier for maintenance than writing a complex one from scratch. Secondly, there is more chance to reuse generic simple instrumentations and data processings rather than a complex dedicated one. Thirdly, it is easier to compare and have confidence in results when instrumentation and data processing are the same among studies. In order to do this, since a simulation model is the composition of existing models and new models, the corresponding instrumentation may also be the composition of existing instrumentations with existing data processors and new data processors. Therefore, each

```
<definition name="Composition"
   extends=PeerLifetime(peer1),
           PeerLifetime(peer2)>
</definition>
```

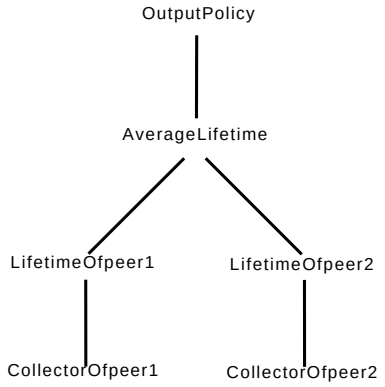

Figure 5: Fractal ADL composition mechanism and the resulting COSMOS design.

simulation model should be accompanied by one or more instrumentations and data processors that could be reused —i.e., at least the instrumentation and the data processors used to validate the simulation model.

4.5 From Real to Virtual System

In OSIF, we use COSMOS to process simulation data. As mentioned in Section 2, it would be appropriate to compare simulation results with the results of an experiment. COSMOS has originally been developed to manage context data in ubiquitous applications [3, 4, 15][2]. So, OSIF can naturally be used for instrumenting and processing output for both real applications and simulations of real applications. As a consequence, the validation of simulation models is much more effective and less buggy, and the level of confidence in the validation process increases.

5. RELATED WORKS

Although some authors carefully describe the implementation details of a simulator and classical discrete-event simulation algorithms (e.g. Banks et al. in [2, 1], or Fujimoto in [8]), none do actually describe and discuss the issues related to the management of the data produced during a simulation run: most of them simply assume that statistics are *computed* during the simulation and either saved on-the-fly for later processing, or directly used to produce a final execution report at the end of each run. Some authors, like Andradóttir [1], propose techniques to reduce the computational complexity of this dynamic observation and on-line statistics computation.

Others, like Himmelspach *et al.* [10], while still mainly focusing on experiment planning issues, acknowledge that handling the huge amount of data produced by a simulation, especially in a distributed environment, is a complex task. For this purpose, they propose a simple architecture in which *intrumenters* instantiate *observers*, that, in turn, may use *mediators* to handle the transmission during the simulation of the data across the network, to their storage destination. In JamesII, the model must notify observers that variables have changed. This prevents reuse without source modification of model that was not originally made for JamesII.

In [9], Gulyas and Kozsik address the issue of separation of concerns in simulation using AOP. But their application iof the AOP paradigm is limited to the gathering of simulation data. They do not consider using AOP for instrumentation and analysis.

In [17], Varga and Hornig address the issue of results' analysis. They propose Scave, a tool to post-analyze simulation data. Scave can apply a batch of analysis to several simulation data files. This favors the comparison between similar studies by using the same analysis process on several simulation outputs but does not raise questions about data gathering.

6. CONCLUSIONS AND PERSPECTIVES

OSIF is a base for creating, analysing and validating instrumentations. OSIF is based on several mature software engineering techniques and frameworks, such as COSMOS, Fractal and its ADL, and AOP. Benefits of OSIF are multiple: (i) OSIF allows a complete separation of concerns between modeling, instrumentation and data processing; (ii) OSIF favors validation results by allowing the sharing of analysis between the real system and the simulated system; (iii) OSIF allows to manage and optimize the flow of simulation data whether we want to live analyze or post analyze simulation data; and (iv) OSIF allows to design and compose complex instrumentations and data processings in a simple way. OSIF has been successfully tested on the OSA simulator through the large-scale simulation presented in Section 3. OSIF has been designed so that there are no connection between the simulator and instrumentation and data processing. Thus, OSIF can be used and reused on any simulator. Since AOP is available for most programming languages, OSIF could be used regardless the simulator and the language used. COSMOS collectors are written in Java, but there are already several ways to integrate non-Java languages, for example, using JNI. The next step to the success is to unite a community around OSIF to build and share COSMOS components in order to enrich the experience of the end-users.

To enrich the OSIF experience, we are planning future works. The first one is derived from the fact that extension and redefinition mechanisms of Fractal ADL can lead to unintended results because of side effects being difficult to predict. A tool for describing, analyzing and verifying data processing as one currently developed by COSMOS (COSMOS DSL) will retain the advantages while avoiding disadvantages. Then, a medium term project is to build on top of the COSMOS instrumentation policy a tool to drive simulation experiments. We plan to bring into play the principles that Himmelspach explained in [10]. Indeed, COSMOS offers all we need to drive simulation such as controlling the number of runs necessary to obtain the expected confidence intervals or automatically cut the beginning of a simulation or refining inputs to obtain the best inputs combination for a study.

[2]See also the following projects: Cappucino on mobile commerce (`http://www.cappucino.fr/`), and Totem on pervasive gaming.

7. ACKNOWLEDGMENTS

This work was partially funded by the European project IST/FET AEOLUS, the ANR project SPREADS and the ARC INRIA project BROCCOLI.

8. REFERENCES

[1] S. Andradóttir. *Handbook of Simulation*, chapter 9-Simulation Optimization, pages 307–334. EMP & Wiley, 1998.

[2] J. Banks, J. S. Carson II, B. L. Nelson, and D. M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, 5th edition, 2009.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and its Support in Java. In *Proc. 7th International Symposium on Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22, May 2004.

[4] 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*, 36(11):1257–1284, Sept. 2006.

[5] D. Conan, R. Rouvoy, and L. Seinturier. Scalable Processing of Context Information with COSMOS. In J. Indulska and K. Raymonds, editors, *Proc. 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, volume 4531 of *Lecture Notes in Computer Science*, pages 210–224, Paphos, Cyprus, June 2007. Springer-Verlag.

[6] O. Dalle. OSA: an Open Component-based Architecture for Discrete-Event Simulation. In *20th European Conference on Modeling and Simulation (ECMS)*, pages 253–259, Bonn, Germany, May 2006.

[7] O. Dalle and C. Mrabet. An instrumentation framework for component-based simulations based on the separation of concerns paradigm. In *Proc. of 6th EUROSIM Congress (EUROSIM2007)*, Ljubljana, Slovenia, September 9-13 2007.

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

[9] L. Gulyas and T. Kozsik. The Use of Aspect-Oriented Programming in Scientific Simulations. In *Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, Estonia*. Citeseer, 1999.

[10] J. Himmelspach, R. Ewald, and A. M. Uhrmacher. A flexible and scalable experimentation layer. In S. Mason, R. Hill, L. Mönch, O. Rose, T. Jefferson, and J. Fowler, editors, *Proc, of the 2008 Winter Simulation Conference (WSC'08)*, pages 827–835, Miami, FL, Dec. 2008.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[12] A. Law. *Simulation Modeling and Analysis*. McGraw-Hill, 2006.

[13] M. Leclercq, A. Özcan, V. Quéma, and J.-B. Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *Proc. 29th ACM International Conference on Software Engineering*, (USA), May 2007.

[14] M. Leclercq, V. Quéma, and J.-B. Stefani. DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9), Sept. 2005.

[15] D. Romero, R. Rouvoy, S. Chabridon, D. Conan, N. Pessemier, and N. Seinturier. *Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments*. Chapman and Hall/CRC, 2009.

[16] R. Rouvoy, D. Conan, and L. Seinturier. Software Architecture Patterns for a Context Processing Middleware Framework. *IEEE Distributed Systems Online*, 9(6), June 2008.

[17] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[18] B. P. Zeigler. *Theory of Modelling and Simulation*. Krieger Publishing Co., Inc., Melbourne, FL, USA, 1984.