UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de
Docteur en Sciences
de l'Université de Nice - Sophia Antipolis
Mention : **INFORMATIQUE**

Présentée et soutenue par
Judicaël RIBAULT

Reuse and Scalability in Modeling and Simulation Software Engineering

Thèse dirigée par : **Jean-Claude BERMOND** et **Olivier DALLE**
préparée au sein du projet MASCOTTE - INRIA / I3S (CNRS/UNS)

Rapporteurs :	Lionel SEINTURIER	-	Professeur - Université Lille 1
	Adelinde UHRMACHER	-	Professeur - Université Rostock, Allemagne
	Gabriel WAINER	-	Professeur - Université Carleton, Canada
Directeurs :	Jean-Claude BERMOND	-	Directeur de Recherche - Laboratoire I3S, Sophia Antipolis
	Olivier DALLE	-	Maître de Conférences - Université Nice Sophia Antipolis
Examinateurs :	Denis CONAN	-	Maître de Conférences - Télécom SudParis
	David HILL	-	Professeur - Université Clermont-Ferrand 2
Président :	Michel RIVEILL	-	Professeur - Université Nice Sophia Antipolis

Acknowledgments

Last thing to do :-)

Résumé

L'étude d'un système à l'aide de simulations informatiques à événements discrets implique plusieurs activités: spécification du modèle conceptuel, description de l'architecture logicielle du modèle, développement des logiciels, scénarisation de la simulation, instrumentation, planification d'expérimentation, configuration des ressources de calcul, exécution, post-traitement et analyse, validation et de vérification (V&V). De nombreux éléments logiciels sont requis pour remplir toutes ces activités. Toutefois, il est fréquent de créer un nouveau simulateur à partir de rien quand on commence une étude à l'aide de simulation. Dans ce cas il est nécessaire de développer de multiples outils prenant en charge les activités de la simulation.

Cette thèse aborde le défi de la création de nouveaux simulateurs tout en réutilisant des modèles et des outils provenant d'autres simulateurs. En effet, la réutilisation de logiciel augmente la fiabilité, est moins sujette aux erreurs, permet une meilleure utilisation des expertises complémentaires, améliore la conformité aux normes, et accélère le développement. La réutilisation de logiciels peut être appliquée à toutes les activités de la simulation. Plusieurs problèmes doivent être résolus pour tirer pleinement profit de la réutilisation. Dans cette thèse, nous abordons trois questions principales: Tout d'abord, nous étudions les solutions pratiques de réutilisation permettant de combiner un ensemble choisi d'éléments logiciels utiles pour la modélisation et la simulation, en incluant aussi bien les modèles, les moteurs de simulation, les algorithmes et les outils; Deuxièmement, nous nous concentrons sur les questions liées à l'instrumentation; Troisièmement, nous étudions le problème de l'intégration d'éléments logiciels provenant d'autres simulateurs dans un nouveau simulateur.

Pour atteindre ces objectifs, nous étudions des techniques avancées de du génie logiciel, tels que le génie logiciel à base de composants (CBSE) et la programmation orientée aspect, sur lesquels nous construisons une solution originale pour la modélisation et la simulation à l'aide de multiples couches réutilisables. Nous avons développé un prototype d'architecture logicielle qui prove la faisabilité de cette solution.

Mots clés: simulation, évènements discrets, aspects, séparation des préoccupations, instrumentation, modélisation, composant, simulation distribuée, réutilisation

Abstract

Studying a system using discrete-event computer simulations implies several activities: conceptual model specification, software model architecture description, software development, simulation scenario, instrumentation, experimentation planning, computational resources configuration, execution, post-processing and analysis, validation and verification (V&V). Many software are required to complete all these activities. However, it is common practice to create a simulator from scratch when starting a new a simulation study. It is therefore necessary to redevelop a whole suite of tools to ensure support for all simulation activities.

This thesis addresses the challenge of developing new simulators that reuse existing models and simulator parts. Indeed, reusing software increases dependability, is less error prone, makes better use of complementary expertises, improves standards compliance, and accelerates development. Reusing software can be applied to all simulation activities. Several problems have to be solved to derive full benefit of reuse. In this thesis, we address three major issues: Firstly, we investigate practical means of reusing and combining valuable pieces of modeling and simulation software at large, including models, simulation engines and algorithms, and supporting tools for the modeling and simulation methodology; Secondly, we focus on issues related to instrumentation; Thirdly, we focus on problems of integration of existing simulation tools.

To achieve these objectives, we investigate advanced software engineering techniques such as component-based software engineering (CBSE) and aspect-oriented programming (AOP), and use them to derive a novel approach for Modeling & Simulation based on reusable layers. We developed a prototype software architecture that proves the feasibility of this layered approach.

Keywords: simulation, discrete events, aspects, separation of concerns, instrumentation, modeling, component, distributed simulation, reuse

Contents

List of Figures

List of Listings

Chapter 1

Introduction

Contents

It is well known in M&S that in order to achieve high credibility of M&S results it is mandatory to take into account the validity of models and simulation studies [L'Ecuyer, 1990, Johnson, 2002, Balci, 2003, Troitzsch, 2004, Kelton and Law, 2000, Sargent, 2008]. But we miss evidence that software engineering of M&S products is taken equally important – although the results achieved as well as their quality may heavily depend on the implementations used. This lack of importance given to the simulation software in the quality assessment of simulation results can be found through recent studies showing that many publications do not mention the information necessary to ensure the replayability of simulations and therefore the credibility of simulations results. In [Pawlikowski et al., 2002], the authors surveyed over 2200 publications on telecommunication networks in proceedings of the IEEE INFOCOM and such journals as the IEEE Transactions on Communications, the IEEE/ACM Transactions on Networking, and the Performance Evaluation Journal. Their conclusion was that "[...]the majority of recently published results of simulation studies do not satisfy the basic criteria of credibility". Similarly, in [Kurkowski et al., 2005] Kurkowski et al. surveyed the results of MANET simulation studies published between 2000 and 2005 in the ACM MobiHoc Symposium : 75% of these papers used simulations, but they found out that less than 15% only of the simulations were repeatable and only 7% addressed such important issues as initialization bias. Thus, it is important to improve the practices. In this thesis we choose a pragmatic approach based on the following claim: without any real incentive, or simply by ignorance of better practices, practitioners will not spontaneously change their habits. Therefore, our tentative answer to solve the quality issue is to improve practices with no additional effort for practitioners, by providing tools that are designed to better support the simulation methodology.

Creating a classic simulator able to achieve quality results already involves a significant amount of work and time. On one hand, a common practice to save time is to reuse existing simulators to conduct new simulation studies. Furthermore, software reuse is known to increase dependability, reduce process risk, makes efficient use of specialists, comply with standards, and accelerate development [Sommerville, 2007]. Notice also that reuse can take many forms. The form of reuse we consider in this thesis is the reuse of existing software without modifications on the source code. Indeed, to capitalize on benefits such as allowing the comparison between studies or avoiding to re-enter the verification process of a model, elements of reuse should not be subject to modification. On the other hand, despite the ever increasing number of simulators that can be used, numerous studies still continue to be conducted using new simulators created ab nihilo. Indeed, the time spent to learn an existing simulator is often believed to be equivalent to the time spent to create a specific simulator that better matches the modeler expectations. However, key activities for achieving quality results, such as VV&A or results analysis, are often neglected in this time evaluation. This leads to results that are difficult to reproduce and whose quality may be questioned, as pointed out in the papers cited above. However, it seems not reasonable to expect that practitioners will change their habits. Based on this observation, our goal becomes to build a framework or software architecture for simulation that helps users to build their own simulation environment, so they can stick to their habits, but still comes

with strong incentives for reuse, so they can benefit from the quality improvements.

The reuse in modeling and simulation can be applied to different activities of the simulation, such as the specification of conceptual models with, for example, the reuse of formal language such as DEVS; the description of the software architecture of the model with, for example, the reuse of an architecture description language (ADL); the development of simulation software such as the reuse of implementation of model, engine, generator or queue; the scripting of the simulation with, for example, the reuse of all or parts of existing scenarios; the instrumentation of the simulation with, for example, the reuse of all or parts of existing instrumentations; the reuse of tools for automated sequential analysis in stochastic simulations, like Akaroa[Pawlikowski and Yau, 1993] ; the configuration of computational resources with, for example, the reuse of a deployment plan; the control of the execution with, for example, the reuse of tools (which allow to stop the simulation if it exceeds a threshold, etc.). the post-processing and analysis with, for example, the reuse of data processing and tools such as Scave; the validation and verification with, for example, the reuse of tests and formal verification process.

Several techniques exist to promote the reuse without modification among which the use of libraries such as SSJ [L'Ecuyer et al., 2002]. SSJ allows to build a simulation focusing on the modeling part by reusing simulation engine and tools that have followed a process of VV&A and are known for their quality. The use of components allows to separate the various businesses of the simulation and allows the experts to work on their own concerns. The use of a middleware such as HLA allows the composition of several simulations in a global simulation. Despite these tools and techniques, many simulation elements are not thought to be reused and it is difficult to integrate them without modification.

Thus, there are typically mixed concerns:

- modeling elements are intertwined which leads to the hyper-spaghettis phenomenon described by Webster[Webster, 1995] in which many connections exist between the software components. Besides the difficulty it adds to understanding and debugging the code, it limits the reuse.

- in component-based models, the structure of the elements of the system (its topology) needs to be described along with their behavior. Mixing these two concerns is a common practice that prevents their independent reuse.

- an instrumentation is required to observe the behavior of the system under study during the simulation and collect data in order to process them (to produce statistics, animation, or any result). The probes that observes and collects data from the simulation during its execution are often mixed with other concerns (such as the modeling concern), which requires to edit the model when we want to change the variables to be observed.

- and many other concerns, such as dealing with the System dynamics, distributed execution and deployment, debugging, verification, and so on.

This list is not exhaustive because it is always possible to add new concerns, and each concern can be divided into sub-concerns. For example, a model may be obtained

by composing sub-models, a common practice found in component-oriented hierarchical modeling formalisms such as DEVS[Zeigler et al., 2000].

This thesis proposes to develop techniques and tools for achieving reuse and scalability in modeling and simulation software engineering. Section 1.1 presents objectives while section 1.2 presents the organization of this thesis.

1.1 Objectives

This thesis aims to provide new solutions for reuse in the field of modeling and simulation. Because many people develop simulations ab-nihilo, our goal is to allow them to do so while having the ability to reuse what exists as much as possible, and at the same time make it reusable. In order to achieve this goal, we must find a way of separating the various concerns found in a simulation software. For this purpose, we aim at investigating the use of advanced techniques of software engineering and programming and implement them in a prototype M&S software architecture as a proof-of-concept. Our solution uses a fully layered approach, which allows to strictly separate modeling and simulation concerns. We validate our technique through the development of two proofs of concepts:

- Open Simulation Architecture (OSA) is our simulation platform promoting reuse and integration of external software elements. OSA is a component-based application based on a layered approach that allows to completely separate concerns and thus encourage reuse and sharing. OSA uses AOP to allow communication between layers that would need to communicate such that the instrumentation layer and the modeling layer. To ensure replayability, OSA uses the Apache Maven project management tool.

- Open Simulation Instrumentation Framework (OSIF) which is an instrumentation framework that can be plugged to any systems and models without modification on them. OSIF allows to share, reuse and integrate analysis software elements. OSIF is based on the same principle of separation of concerns as OSA and shows that all the techniques for separation of concerns can also apply to a third-party tools.

While reuse is important for modeling and simulation, it should not come at the expense of performance. OSA and OSIF have been developed keeping in mind that the techniques used should not degrade the performance significantly, and should instead serve as a basis for implementing high performance distribution and parallelization mechanisms such as optimistic or conservative parallelization and parallelization of executions following an experimental plan.

1.2 Dissertation Roadmap

The remainder of this document is organized into six parts outlined briefly in the following paragraphs. Chapter 2 focuses on the state of the art relating to our work. Chapter 3 introduces design considerations for M&S software. Chapter 4 describes our contributions

to the process of building a reusable and open simulation architecture. Chapter 5 describes our contributions to the problematic of instrumentation in M&S. Chapter 6 describes our contributions to the problematic of integrating third-party tools from other simulators. Chapter 7 presents the experiments we conducted to validate our proposals.

Design considerations In this chapter, we consider design questions for M&S software. We claim that discussing, designing, developing, and comparing M&S products should start with software engineering considerations. We shortly introduce some of these engineering concepts and discuss how these relate to the M&S domain.

State of the art In this chapter, we start with a survey of the solutions actually used in existing simulators to promote separation of concerns and reuse. Then, we introduce technologies used in this thesis such as component-base software engineering concepts or aspect-oriented programming. We present the state of the art of component models used in software engineering and explain why we chose the FRACTAL component model and related tools. Then, we present M&S concept and terminology used in this thesis, and we finish by presenting open questions on instrumentation and integration.

Contributions to Reuse In this chapter, we investigate practical means of reusing and combining any valuable piece of M&S software at large, including models, simulation engines and algorithms, and supporting tools for the M&S methodology. Then, we focus on how to provide distributed executions means that require no modification on simulation software as well as models. We also present our solution to develop and reuse models, scenarios and engines using aspect-oriented programming and component models such as the FRACTAL Component Model (FCM) through the OSA architecture.

Contributions to Instrumentation In this chapter, we study mechanisms to implement efficient and non intrusive instrumentation of simulation models, without changing or interfering with the code of the model. 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 chapter, we address issues related to the management of these data and their online processing, such as: (1) the mixing of the instrumentation code with the modeling code; (2) the amount of data to be stored. It may be enormous, and often, a significant part of these data are useless while their collection may consume a significant amount of the computing resources; and (3) the difficulty of comparing studies since each user (model developer) builds its own ad-hoc instrumentation and data processing. Last, we present OSIF, a 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.

Thoughts About Integration In this chapter, we consider the problem of reusing parts of existing simulators in a new one. We started from the observation that despite no single simulation software seems to be perfect, most of the elements required to make a perfect simulator already exist as part of existing simulators. This chapter presents our solution to integrate existing simulation elements such as models and engines thanks to the ability of FRACTAL to encapsulate softwares. It is also interesting to integrate a simulator as a back-end, particularly to reuse existing experimental planning. To demonstrate the feasibility of such integration, we use our demonstration platform OSA as front-end for the integration of existing models and engines from another simulator, but also as a back-end for integration of existing experimental planning from another simulator.

Application and Performances In this chapter, we present results we have obtained through various simulation experiments with original features. We present the model to simulate, and the objectives in terms of scale, deployment, execution time and methodologies. The simulated system is a data storage system running on a P2P overlay network. We show that this model scales very well. Then, we present the results obtained and give a quantitative assessment in terms of reuse, instrumentation and integration.

Chapter 2

State of the Art

Contents

Since the early age of programming, people have sought for reuse. Reuse applies to several domains of software engineering such as specification, architecture, data, source code, design, documentation, templates, human interface, plans, requirement, or test cases ([Barns and Bollinger, 1991], [Jones, 1993]). From a historical perspective, we generally attribute reuse as a discipline of software engineering to McIlroy [McIlroy et al., 1969]. Indeed, he is the first to propose a formal reuse approach.

In this Chapter, we present the state of the art about reuse in software engineering and reuse in modeling and simulation. Section 2.1 defines reuse. Then, section 2.2 presents reuse in the general field of software engineering Section 2.3 presents reuse in the modeling and simulation field Section 2.4 presents tools and techniques used in this thesis. Finally, section 2.5 presents a discussion on reuse using the example of Eclipse.

2.1 Definition

In this thesis, we agree with the definition described in [Krueger, 1992]: reuse is "the process of creating software systems from existing software rather than building them from scratch". Reuse can sometimes require changes to match the need. For [Lim, 1994], all modifications must be prohibited since modified elements are no longer the same. For Cooper in [John, 1994], modification could be done, benefits from reusing and modifying is greater than not reusing at all. To McIlroy, reuse could require some modification, but in that case modification must be made carefully. We believe modifications should be applied only if they don't change the specifications of the software component. In component-based software engineering, these specifications define the functionnal, or business part of a component. Hence we define component reuse as the ability to use a component in a different context without changing its functional part.

2.2 Reuse in Software Engineering

First, section 2.2.1 presents reuse motivations, benefits and drawbacks. Section 2.2.2 presents metrics and models used to measure reuse performance. Section 2.2.3 presents reusing techniques.

2.2.1 Motivations and Benefits

As in the hardware industry, the software industry is asked to build more complex and powerful software and to put them on the market quickly. Maximizing the reuse of code (tested, checked or certified) would minimize the development of new code and therefore lower the development cost, time spent and improve the quality of software. Studies ([Tracz, 1988] and [Mili et al., 1995]) show that a significant portion of code can be reused from one application to another.

The benefits of reuse have been the subject of numerous publications ([Taivalsaari and Jyväskylän, 1993], [Schäfer et al., 1993], [John, 1994], [Mili et al., 1995], [Karlsson, 1995], [Sametinger, 1997], [Sommerville, 2007]). In [Sametinger, 1997],

Sametinger summarizes the qualitative and quantitative effects of reusing as follows:
Qualitative effects:

- **Fewer bugs** Reusing code involve more feedback and thus more fixed bug that finally improve the quality and reliability of software.
- **Productivity improved** Although the introduction of systematic reuse can have a negative impact on productivity in the beginning, long term systematic reuse improve productivity.
- **Lack of performance** Generic components that can be used in several softwares can result in a lack of performance compared to a dedicated code optimized for a special application.
- **Interoperability** Applications that share the same component are made de facto interoperable if this shared component allows them to communicate.

Quantitative effects:

- **Less code to write** and thus less time spent for developing new softwares.
- **Time to market is shortened** due to the reduction of development time.
- **Less documentation to write** Indeed, reusing software element means that we could also reuse dedicated documentation. This imply to write detailed documentation when developing new reusable software element and thus spend time there.
- **Less maintenance** Must be maintained only the newly created code. The reused code is maintained by a third group. However updated components can be expensive.
- **Additional training costs** An additional cost to train engineers to handle reusable components must be took into account, but that cost is quickly amortized long-term.
- **Small team** The number of people necessary for the production of software no longer needs to be as important. It results a better communication between team members and thus an increased the productivity.
- **Quick prototyping** due to the assembly of existing component.

2.2.2 Metrics

Benefits and drawbacks of reusing vary depending on the project, time and money we have to invest, and life span of the project. To determine whether reuse is beneficial and worth to be put in place, a number of metrics have to be taken into account. In [Frakes and Terry, 1996], authors survey metrics and models for software reuse as summarized hereafter.

Reuse cost-benefits models attempt to measure and predict the quality and return on investment introduced by reuse. Several similar models have been proposed ([Barnes et al., 1988], [Gaffney Jr and Durek, 1989], [Poulin et al., 1993]). It appears from these models that costs rise quickly with component size and complexity [Favaro, 1991]. Thus, to improve the quality of investment, we can act on 3 lever: "increase level of reuse, reduce the average cost or reuse, reduce the investment needed to achieve a given reuse benefit".

In [Margono and Rhoads, 1992], authors show that the cost of development for a reusable component is more than the cost of development for an equivalent non-reusable component. However, In [Frakes et al., 1991] and [Chen and Lee, 1993], authors show that reuse improves productivity. The reuse cost-benefits models measure and help to decide when it will be profitable to develop a reusable component depending on the number of times we expect to reuse it. If the objective is not the cost but the quality, several studies ([Card et al., 1986], [Browne et al., 1990], [Frakes et al., 1991], [Agresti and Evanco, 1992] and [Mohagheghi and Conradi, 2008]) show that the reuse is beneficial to the quality of software product.

Reuse maturity models attempts to measure the systematically reuse ability of an organization and identify issues that prevented them to apply systematic reuse. Models were proposed by [Koltun and Hudson, 1991] and [Davis, 1993].

Amount of reuse metrics focus on the percentage of reuse in the life cycle of an object to measure reuse rate. It is based on the number of reused lines versus the total number of lines of code. However, the reuse can take place at several locations and at different levels of granularity ranging from the reuse of source code to the reuse of external tools or services. In this case it is necessary to find other metrics to measure the rate of reuse. In [Frakes and Terry, 1994], Frakes and Terry propose the concept of "reuse level" to calculate the overall reuse rate of a project depending on the reuse rate of each level. In [Karunanithi and Bieman, 1993] and [Chidamber and Kemerer, 1994], authors provides metrics specially adapted for measuring reuse rate in the context of object-oriented system. In [Washizaki et al., 2003], authors provides a metrics suite for measuring the reusability of black-box components without any source code.

Reuse failure modes model In [Frakes and Fox, 1996] and [Morisio et al., 2002], authors allow to identify and classify barriers to reuse. There are many reasons to fail to establish systematic reuse process. Studies conducted on this subject show that one of the main reasons is that people do not even try to implement systematic reuse, and this may be due to the small amount of resources allocated for reuse. In academic research, many studies are made from scratch and we can think that people do not even think about trying to reuse. This reflects the fact that teams are not programming experts. Indeed, research teams are generally small independent team, without big organization around to manage project. In the end those who reuse are the teams that develop solutions to encourage reuse. Others failure reasons are (in order of importance): inability to reuse, impossible to

understand the component, the component is not valid or effective, the component does not exist, ...

Reusability metrics can measure the rate of reuse of an object. It also helps to identify what are the objects that would benefit from being reusable. In [Selby, 1989], Selby after conducting a study on NASA's codes identified that codes most reused shared common characteristics. In [Basili et al., 1990] and [Dunn and Knight, 1991], the authors propose an approach to identify modules loosely coupled. These modules do not require much effort to be reusable. Finally, authors target simple modules, focused, well-documented and independent from other modules as good candidates for reuse.

Reuse library metrics address the problem of storing and searching for reusable components. In [Frakes and Gandel, 1990], Frakes and Gandel define a reuse library as " a repository for storing reusable assets, plus an interface for searching the repository". Authors also identified most common classification schemes such as enumerated, faceted, and free text indexing. They show that the way classification is done is not so important. The most important things is to be able to determine the quality of the assets to reuse. Authors indexing schemes evaluation depend on cost, searching effectiveness, support for understanding, and efficiency. In [Frakes and Nejmeh, 1986], Frakes and Nejmeh propose some metrics indicator that must be visible in a reuse library. Among those indicators, we find the rate of successful reuse. Today with Internet and the Web 2.0, we rely a lot on the opinions and commentary of a community to know what are the elements to be reused or avoided.

2.2.3 Techniques

In [Sametinger, 1997], Sameting classify reuse techniques into 2 categories: "compositional reuse" and "generative reuse".

Compositional reuse The compositional reuse is the complex assembly of simple software element. New software elements are developed in case none match.

There are many different ways to assemble the components, but there must be a standard to allow components to communicate. For example, component models such as Corba, Java Beans, EJB, DCOM and modern composition systems, such as aspect-oriented programming (Aspect-J). Looking back in the past, [Kernighan, 1984] used to define the assembly of components (in Unix) as connecting the output of a component with the input of another component. Other techniques have emerged since then. Compositional reuse is the composition of fragments from existing software as part of new software development. This can be done in an ad-hoc and unsystematic way such as reusing portion of source code, or in a more structured way, for example by clearly expliciting interfaces [Röhl and Uhrmacher, 2008] and contracts [Chang and Collet, 2007]. The composition of Web services to create mashups is another modern form of component reuse. At a coarse grain level, complete systems are available for reuse as "virtual appliance", i.e. virtual

machine images specialized for a particular service; in that case the composition method is standard networking, and the network itself can be a virtualized.

To reuse components, we must be able to identify, classify, rate and find components as well as their documentation. Moreover, because of the nature of software engineering (almost everything is possible compared to the hardware where there is more constraint), customers do not want to settle for generic software and want specialization, so there must be compositional mechanisms but also specialization and adaptation mechanisms. All reusable items should be available in a repository (also called library). A component repository is a database for the storage and retrieval of reusable components. In [Moore, 1994], Moore classifies repositories in 3 categories:

- local repository which contains usually used components.
- domain-specific repository which contains components specific to a domain or business.
- reference repository referencing all the components. It acts as a general directory and index of deposits.

The "reuse library metrics" already mentioned in section 2.2.2 address the problem of storing and searching for reusable components. Indeed, the functions of identification, classification and research in deposits are used to find the ideal candidate for reuse. These features must be taken very seriously and are the source of actual research.

Generative reuse Generative reuse is based on the reuse of process generation. For example, the use of Lex and Yacc allows the generation of syntactic analyzer from a description. More recently, we could cite the Eclipse Modeling Framework or the Hobo framework for Ruby on Rails[Dall et al., 2010]. We can also see programming languages as low-level generator (conversion into machine code). Generative reuse are dedicated to a specific domain and allow to reuse only source (specification, source code, template, ...).

Elements of reuse There are many elements involved in software development that we might want to reuse:

- **Algorithm** We reuse algorithms concepts and ideas we find in several sources (books, internet). The reuse of algorithms often requires changes to adapt to the language or the data on which we want to implement it.
- **Function library** One of the most reused element.
- **Class library** The object version of the function library. Reusability benefits greatly from the concept of inheritance, polymorphism and dynamic bindings. Cons, classes works through family, 2 family will not declare the same interface and thus will not be able to be interchangeable.

- **Software architecture and design** We can reuse patterns of well known software architecture: communication processes, hierarchical layers, pipes and filters, clients and servers, interpreters [Garlan and Shaw, 1994]. The reuse of existing design is primarily the reuse of knowledge due to experience. We can reuse the design of complete system or sub system, interfaces, implementation, data structure, algorithms.
- **Framework Class** Reusing a framework allows to reuse all non-functional aspects done by experts and thus allows the developer to work simply on the functional part. It extended the principle of reuse class library applied to large-scale application.
- **Design pattern** Reusing design patterns allow to apply well-known and identified programming models to avoid common errors of design. This puts the emphasis on reuse at the design level, but also allows to understand design and code faster.
- **Business process** Reusing business processes using standards such as BPEL (by OASIS) or BPMN (by OMG), allows to reduce cost from the business changing and improve productivity.
- **Application** Reusing applications allows to focus on the development of the environment and let users choose what tools they want to use.
- **Documentation** It is possible to reuse documentations in conjunction with the reuse of software elements.

It is also possible to mix both techniques. By generating reusable component, or by composing template used in the generative process.

Component Frameworks The general idea of a software component is not so new (see [McIlroy et al., 1969]), but one had to wait for almost 30 years and approaches such as EJB [Bodoff et al., 2002] before industrial strength component frameworks (CF) became available. Since then, many component framework have been proposed such as ArchJava [Aldrich et al., 2002], PECOS [Genssler, 2002], K-Component [Dowling and Cahill, 2001], OpenCOM [Clarke et al., 2001], OSGi [osg, 2004] or FRACTAL [Bruneton et al., 2006]. A comprehensive study of the domain produces more than 20 different frameworks. Some of these component framework are general purpose, whereas others wish to target specific application domains such as realtime systems, or dynamic environments. The high level concepts of these frameworks are similar. They all deal with the idea that "*a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*" [Szyperski, 2002].

Architecture Description Languages Architecture Description Languages (ADL) are often used as a complementary paradigm to components. The idea is to declaratively describe the connections between components and their assembling. As for component framework, many ADL have been proposed in the literature. Readers can refer to [Medvidovic and Taylor, 2002] for a survey of the domain. The architecture description

languages are intended to formally describe software architectures of a computer system. An architecture description language fits perfectly to component-based system as it specifies interactions between components. "A software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [Bass et al., 2003]. Concepts commonly associated with the field of software architectures are [Medvidovic and Taylor, 2002]:

- components that represent the basic entities of an application,
- connectors that identify the types of interactions between components of the architecture,
- configurations that describe an architecture in terms of components and connectors,
- composites that reify a configuration as a component.

Lau and Wang's Survey of Component Frameworks Component models defined for applications are not necessarily suitable for the development of modeling and simulation platforms. In particular, modeling and simulation platforms are looking for models of components that are efficient light enough not to penalize the conduct of the simulation. But there are models of components that enjoys the benefits of programming components while minimizing its disadvantages. These models of lightweight components do not support every technical properties of the components (e.g. transactions and security) but offer more flexible solutions to suit the needs of the container component. The container is the runtime support of a software component and provides various services implanted by the software platform. The support component is completed in order to introspect and reconfigure applications dynamically. Moreover, models of lightweight components used increasingly notions of hierarchy and division in order to increase the modularity of the systems.

In [Lau and Wang, 2005], Lau and Wang survey existing software component models. Rather than discussing them in some random sequential order, they survey the models in meaningful groups. They present four categories that cover models accordingly to their compositions in an ideal life cycle.

An idealized component life cycle is represented by a three-step phase: the design phase, the deployment phase and the runtime phase. In the design phase, "a component is designed and implemented in source code, by a component developer". In the deployment phase, "a component is a binary, ready to be deployed into an application by a system developer". In the runtime phase, "a component instance is created from the binary component and the instantiated component runs in a system"

In category 3, there is no possibility to store into the repository composite components. In categories 1, 2 and 4: the composition of the component instances (in the runtime phase) is the same as that of the components in the design phase. Thus, the assembly is done during the design phase. Category 1 switch from implementation to instantiation without any repository whereas categories 2 and 4 store the result of the composition into the

repository. Difference between categories 2 and 4 is that in category 2 it is not possible to reference an existing component stored in the repository during the design phase whereas it is possible in category 4.

From a simulation point of view, it is interesting to identify these phases in order to know what kind of component model corresponds to our objectives. For replayability, it is important to store models (primitive or composite) in a repository. For reusability, it is important that in the design phase we can use model (primitive or composite) stored in the repository. The only category that can store models in a repository but also use them in the design phase are the component models of category 4.

Lau and Wang identify in category 4 Koala [Van Ommering et al., 2002], SOFA [Bures et al., 2006] and Kobra [Atkinson et al., 2000]. FRACTAL though placed in Category 1 uses a repository (Maven) for archiving models. These models can be referenced from the design phase so I would place also FRACTAL as a component models of category 4.

2.3 Reuse in Modeling and Simulation

This section presents reuse in modeling and simulation software engineering. First, section 2.3.1 provides a background on modeling and simulation and presents the terminology used in this thesis. Section 2.3.2 presents several modeling and simulation software. Section 2.3.3 presents reusing techniques used by simulation software. Finally, section 2.3.4 presents open questions for reuse about instrumentation and integration of existing software elements.

2.3.1 Background on Modeling and Simulation

The computer simulation has become essential for the design of telecommunication networks, roads, civil and military aviation, robotics, and for the study of natural systems in physics, chemistry and biology, but also human systems in economics, social sciences, strategy and military defense. The advantage of computer simulation is two-fold. On one hand, its rapid implementation gives it a significant economic advantage over a solution based on a real infrastructure. On the other hand, its ability to simulate conditions that are difficult or impossible to create in a real infrastructure is an asset in many areas. The advantages of computer simulation also allow research teams to test their theories.

This section presents necessary knowledge in the modeling and simulation field related to this thesis.

Modeling Before studying a system, it is necessary to target the use of the model. Depending on the objectives and decisions, we can distinguish models to:

- better understand the operation of a system,
- predict the future behavior of a system,

- optimize performance through analysis of alternatives,
- design systems that do not exist,
- develop equipment/structures (circuits, architecture ...),
- rapidly obtain a working model to test ideas and seek advice from stakeholders,
- plan for the future (development of new infrastructures, etc.),
- assist in the acquisition of expensive equipment: can help decide which to buy by analyzing scenarios/alternatives,
- test ideas before moving to production,
- train to react to certain situations (war games, economics, ...),
- have a better understanding of phenomena and mechanisms in science,
- view system (games, animations).

Once we have the objectives clarified and questions the model must meet raised, it is necessary to establish a set of assumptions on the behavior of the system and in relation with the question(s) to solve. All these assumptions (verified or not by experiment) are the conceptual model. The conceptual model is then translated into a physical or mathematical model. A mathematical model "represents a system in terms of logical and quantitative relationships that are then manipulated and changed to see how the model reacts" [Law, 2007]. A mathematical model can be solved using analytical solution or simulation. "If the model is simple enough, it may be possible to work with its relationships and quantities to get an exact, analytical solution . . . [in the other case]. . . the model must be studied by means of simulation" [Law, 2007].

A simulation model is a simplified representation of a system that behaves like the original from a certain point of view. As mentioned before, a model can either be a purely mathematical construct (algebraic, calculus, random number generation, . . .) or has to be represented in a programming languages for more complex systems. In [Law, 2007], Law classified simulation models into those that are static or dynamic, deterministic or stochastic, or continuous or discrete :

- "A static simulation model is a representation of a system at a particular time, or one that may be used to represent a system in which time simply plays no role. . . . a dynamic simulation model represents a system as it evolves over time."

- "If a simulation does not contain any probabilistic (i.e., random) component, it is called deterministic". Otherwise it is called stochastic.

- In a similar way, a discrete simulation model is a simulation model with a finite number of state whereas a continuous simulation model is a simulation model where state variables change continuously over time. Notice that it is not necessary to have a discrete simulation model to simulate a discrete system and vice versa.

Simulation There are several definitions for the simulation terms. In [Law, 2007], Law simply describes simulation as the fact of using a computer to evaluate a model numerically.

A more detailed definition is given by [Banks, 1999] and introduces the notion of system: "Simulation is the imitation of the operation of a real-world process or system over time. Simulation involves the generation of an artificial history of the system, and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system that is represented." The general definition of a system can be borrowed from [Law, 2007]: "a system is defined to be a collection of entities, e.g., people or machines, that act and interact together toward the accomplishment of some logical end." From a M&S point of view, [Zeigler et al., 2000] gave a more formal definition that is widely used in the community: "[A system] is viewed as a source of observable data, in the form of time-indexed trajectories of variables."

A system can be discrete or continuous [Law, 2007]. A continuous system is a system where state variables change continuously over time. There are 2 ways to make transitions between states in discrete systems: at a fixed time period, so called time-driven, or at variable times caused by asynchronous event, so called event-driven. In discrete-event simulation, "each event occurs at an instant in time and marks a change of state in the system" [Robinson, 2004]. Thus, the behavior of a system is orchestrated by a sequence of events played chronologically (according to their date of occurrence).

Simulation activities Studying a system using component based discrete event simulations implies several activities. The first modeling step consists in describing the model the system expert has in mind. This **conceptual model specification** may range from the informal ones (textual, manual drawings) to the more formal ones (DEVS, ...). Then, it is necessary to translate the conceptual model specification into a **software model architecture description**. This description needs to be compliant with the targeted computer simulation software and usually combines a list of components and a topological description of their interactions (bindings). Next, the **software development** consists mainly in implementing the behavior of model components using a programming language. At this step, the model code is ready to be executed.

However, before running simulation experiments, simulation parameters need to be fixed. First, the **simulation scenario** needs to be fully defined by setting up the initial parameters of the components in order to reach the initial state of the system. The model also needs to be **instrumented** with probes that will collect data samples during the simulation run. Previous configuration elements constitute an **experiment plan**: studying a system using computer simulations often turns into comparing the behavior and performances of the considered system using several variations of a same basic scenario. These variations consist in using different values for some of the initial parameters in the basic scenario.

The last step consists in **configuring the computational resources** to deploy the simulation. **Execution control** allows to execute the simulator through different ways such a debug mode, start/stop/resume and so on. When the simulation is completed, a **post-processing and analysis** step prepares the data collected during the simulation runs (merging and formatting the data of several simulation runs) and run computations on these data (e.g. statistical computations, graph plotting, ...). In the end, a **validation**

and verification step verifies that the software model behaves as expected.

Discrete simulation There are two main approaches to construct a discrete simulation, as described by [Banks et al., 2004]:

The event-oriented approach "[...] concerns the modelling of a system as it evolves over time by a representation in which the state variables change instantaneously at separate points in time" [Law, 2007]. We explained in section 2.3.1 that the behavior of a system is orchestrated by a sequence of chronological events. The task of the simulator is to choose the next event in the future event-set and thus advancing the time of the simulation. The usual algorithm is called next-event time advance: "after all state changes have been made at the time corresponding to a particular event, simulated time is advanced to the time of the next event, and that event is executed. Then simulated time is again advanced to the scheduled time of the next event, and the procedure is repeated." [Fishman, 2001] Elements contained in events perform state transition and add new events in the future-event set. Thus, it is possible to advance rapidly in the simulated time when the gap of time between events is important. Conversely, if the gap of time between events is small, the execution time of the simulation will be more important. Unlike the time-driven simulation, the event-driven simulation can move quickly or detail the sequence of execution with great precision according to the scenario being studied.

The process-oriented approach involves multiple threads. "A thread is a separately schedulable unit of execution control, implemented as part of a single executing process" [Banks et al., 2004]. The usual algorithm is to have an additional thread whose roles is to organize the execution of simulation threads. "processing for that thread involves removing the least-time event from the event-list, reanimating the simulation process thread (or threads) associated with that event, and blocking until those threads have completed" [Banks et al., 2004]. A thread can release events during its execution. At the end of its execution, a thread blocks, waiting for an event or terminate. There may be one or more active threads at a time depending on whether one wants a sequential or parallel execution.

It is easier to implement an event-oriented simulator rather than a process-oriented simulator because there is no need to manage threads control which require some expertise in the programming language. Moreover, an event-oriented simulator is generally faster because there is no context switching between processes. On the other hand, event-oriented approach forces the modeler to design and implement more model-management logic to translate a system into a series of states and transitions. Finally, we must choose between the ease of modeling or the ease of developing a simulator (and possibly the execution speed).

Distributed simulation The use of networking technologies has led to the emergence of specific simulations, called distributed simulations possibly involving several different simulators connected by one or more computer networks. In this type of simulation, the interoperability between distributed components is essential to ensure a coherent global behavior. All actors must communicate and interact distributed by following a common

framework which is set by a distributed simulation architecture. There are quite a number of advantages over local simulations described by [Fujimoto, 2000] and reported hereafter:

- Reduced execution time: we can possibly reduce the execution time of the simulation by running a simulation in a distributed manner across multiple computational node in parallel. We can hope to gain a factor equal to the number of computational node that are used if the time used by the communications is negligible compared to computation time. Actually, in some rare cases the speed-up can even be greater than the number of computational nodes, when dividing the computation leads to a better use of processor and cache.

- Geographical distribution: Simulations of virtual reality takes advantage of the distribution by allowing multiple participants that are physically located on different sites to interact with each other.

- The integration of distributed simulation running on different platforms provides a new virtual simulation environment. This is the case in military simulations involving multiple simulation platforms (infantry, aviation, ...) through the HLA standard.

- Fault tolerance: the distribution increase the risk of failure, but on the other side decreases the critical breakdown. Indeed, if a failure should occur on a computational node, others computational nodes can continue to run normally. That's the advantage of batch processing distributed across multiple computational node.

Terminology

Scenario A scenario is in charge of reproducing the environment in which the simulated system is placed. A scenario produces stimuli that are applied to the model in order to influence its behavior in a controlled way. These stimuli are also called "exogeneous events", because they originate from outside of the model.

Simulation (or Computer Simulation) A simulation is a program that is meant to reproduce the behavior of an original "system" (or process) placed in a given context, using a "model" of this system. The context in which the model is placed during a simulation may have its own complex dynamics, following some "scenario". Usually, the behavior of the original system is reproduced to be observed. These "observations" are collected following an "observation policy" and implemented using an "instrumentation". An "experimental framework" may also be used when the simulation goal is to build in silico experiments in which the observations need to be associated with the situations created by the scenario. Often, the dynamics of a simulation is handled by a part of the simulation software called the "simulation engine". The "simulation engine" and other generic parts of a simulation that can be reused for many simulations are usually referred to as a "simulator".

Simulation Run A "simulation run" describes a "simulation" executable with all its "execution parameters", that is ready for execution. A simulation run produces a certain amount of data, called the "simulation run output". The execution parameters explicit details that are not supposed to have effect on the simulation run output, such as where to find the data needed for the simulation or where to save the data files produced by the run. A simulation run can be complete or partial. A complete simulation run is a run that ended normally after the full computation of the simulation. Multiples executions of a complete simulation run should always produce the same data. Note: A partial simulation run is NOT guaranteed to produce a subset of the data produced by the same simulation run once complete.

Simulation Run Output (or Simulation Output) A simulation run output should not depend on the execution context of the simulation run, i.e. a given simulation run executed on two different computer architectures should always produce the same output (inclusive of computational errors). Note: A simulation run output should take into account the computational errors, and any output difference that falls within the expected computational error margin should not be considered significant. Therefore, two different executions of the same run can still produce slightly different raw data, especially when they result from execution that occurred on different architectures.

Simulation Run Execution (or Simulation Execution) A simulation execution describes a particular execution of simulation run including its execution context (computational resources, operating system specifications and libraries). A simulation run execution can be interactive or non interactive (batch). Several executions of the same "simulation run" obtained in the same execution context should always output the exact same raw data. Note: the execution context makes no assumption on the computational speed: two physically distinct computational resources are assumed to be identical as long as they always produce the exact same raw data for any possible simulation run they could be given to execute.

Trajectory An history of the (consecutive) values taken by a variable during a simulation. A trajectory is kind of "observation" that includes timings.

Observation A series of consecutive data samples originating from the same "observation probe". A simulation may generate an arbitrary number of observations. When the observation probe is associated to a variable and the data samples are the successive time-stamped values of this variable, the observation may also be called a trajectory. Note: Some observations produce a single value (e.g. the final value of a counter.)

Observation Sample or Datum Sample (pl. data samples) A datum produced by an observation probe. An observation sample may or may not be time-stamped (with the simulated time at which it was produced). A datum may or may not be a numerical value, e.g. in a network simulation, an observation sample may be a packet header.

Observation Probe or Data Probe Any source that is able to produce data samples during a simulation. A probe maybe active or passive. A passive probe can only return or generate data samples on demand. In addition to the data samples, an active probe is able to produce notifications. The notification mechanism is implementation dependent (trigger/call-back function, observation event, etc) and may or may not contain the data sample.

Observation Policy An observation policy lists the set of probes used in a simulation and defines when and how they produce "observation samples". For an active probe, the observation policy may be as simple as collecting a data sample every time the probe issues a notification. For a passive probe, some logic is needed to decide when to request a value from the probe: e.g. at regular time intervals, or when a given threshold is reached, or any other condition is reached (e.g. the end of simulation).

Instrumentation An instrumentation implements an "observation policy" and defines the on-line computations that are applied to the "observations". Such on-line computations may include statistics computations, filters, buffering, or I/O operations.

Scalability The scalability of a simulation is its ability to use all the resources at its disposal to execute larger models, but also to use these resources to speedup the simulation using, for example, conservative or optimistic approach.

Deployement The deployment of a simulation is its local or remote execution. Parallel deployments can be done to speedup the execution of the experiment plan.

2.3.2 State of the Art in M&S Software

Many survey already exist comparing simulators in network studies. Most of them only focus on the performance and on ease of modeling. We believe that support for the simulation methodology and credibility of simulation results are just as important as the aforementioned point. In [Begg et al., 2006], the following criteria are used for comparison: modeling capabilities, credibility of simulation models, credibility of simulation results, extendability and usability. For their study, the authors compare several simulators such as NS-2 and OMNeT++. These simulators are well known and are used in numerous publications in the field of network simulation. The authors conclude that “none of currently available simulators satisfies all requirements” they need for their studies and it is better to create a new simulator from scratch although it takes a long time in development, validation and verification (the authors refer to a year of development to achieve what they want to get). Their conclusion agree with the subject of this thesis is that many of the actors prefers to rebuild their own simulator knowing what is going to cost them and it is therefore important to study the possibility of reuse.

In this section, we present the state of the art of techniques used in reusing, integration and instrumentation in the field of modeling and simulation.

JAMES II JAMES II [Himmelspach and Uhrmacher, 2009a] is a general and open framework based on the "Plug'n simulate" concept [Himmelspach and Uhrmacher, 2007], which allows developers to integrate their modeling and simulation methodological ideas into, and to create their applications upon an existing framework. The plug-in concept allows to add any number of extensions per extension point, and thus prototypical implementations by researchers or students can coexist with "high end" / sophisticated solutions for practical use. Although JAMES II has been built using a plug-in based architecture, parts of JAMES II have been created using a "service oriented architecture". For the parallel and distributed computation, the core of JAMES II contains classes for a main server, and computation servers. These, as well as data sinks, are treated as services which can be used for the execution of an experiment. In addition JAMES II is split into a front-end, and a back-end (at least on two sites): on one hand JAMES II can be integrated into any other application (front-end), and thus it then forms a back-end of this application. Nevertheless JAMES II ships with an integrated, extensible front-end as well. Extension points allow to add front-end plug-ins for many back-end parts. On the other hand JAMES II supports the differentiation between symbolic (front-) and executable models (back-end). Modelers benefit from the flexibility that the framework provides with respect to modeling, simulation, and analysis methods, supporting effective and efficient simulation studies, and the well tested methods add to the credibility of the results achieved.

The goal of JAMES II is to provide a framework, reusable for a wide range of applications and supporting the needs of different users, from modeling formalisms over simulation algorithms to experimental design, validation, and optimization methods. This concept together with currently more than 400 plug-ins, and an explicit representation and storage of experiments ease developing modeling and simulation methods and contribute to a systematic experimental evaluation of methods.

DEVS B.P. Zeigler defines in [Zeigler, 1976], a formal specification for discrete event systems: DEVS. This formalism has been introduced as a universal abstract formalism independent from the implementation. It may, in its capacity for abstraction, express systems defined in traditional formalisms such as differential equations (continuous time) and differences equations (discrete time). The concept of atomic and coupled models, introduced by [Zeigler, 1984], provides a way to construct composite models, reusing descriptions stored in library.

The atomic model is a non-decomposable element. The behavior of this element is governed by a discrete event model. Fundamentally, an atomic model has a time base, inputs, states, outputs and functions to determine the next states and outputs from current states and inputs.

A coupled model is a structural model. It describes a structure by interconnection of basic models. Each basic model of the coupled model interacts with other models to produce the overall behavior. The basic models are either atomic models or other coupled models, the coupling of these models is carried out hierarchically.

Parallel to the development of DEVS models presented above, B.P. Zeigler has devel-

oped the concept of abstract simulator [Zeigler et al., 2000]. The simulation architecture is derived from the hierarchical structure of DEVS coupled models. An abstract simulator is an algorithmic description to implement the functions of the model to generate its behavior. Such a simulator is obtained by matching for each element of the model a component of the simulator. The construction of a simulator independent from the model allows a separation, during development, of the modeling and simulation parts. To perform a simulation, a hierarchy of processors, equivalent to the hierarchy of models is constructed. Each component of the model is associated with a processor of the hierarchical structure of the simulator. Each processor participates in the simulation by executing the functions that express the behavior of the model.

Processors are: Simulator that provides simulation of atomic models using the DEVS-defined functions; Coordinator that ensures messages routing between coupled models depending by the definitions of coupling; Root Coordinator who manages the overall simulation. Processor starts and stops the simulation and manages the global time. The simulation is carried out through exchange of specific messages [Zeigler et al., 2000] between differents processors.

CD++ CD++ [Wainer, 2002] is a software package for M&S based on the DEVS formalism. CD++ can be executed on one computer or on several computers. It can also be executed in real time or in parallel. Atomic DEVS models can be programmed in C++. CD++ is accompanied by CD++Builder which is an Eclipse plugin providing a development environment for CD++ simulation projects. CD++ allows in addition to parallel executions to interface with other machines or services. Several solutions are available for integration and reuse such as the compatibility with the HLA standard [Pearce, 2003] or through the RESTful protocol, i.e. using webservices. The use of web services enables the integration of application or data from heterogeneous sources (mashup) within the simulation. In [Harzallah et al., 2008], the authors describe how they combined three sources (the CD++ simulator, a worldwide weather service, and Google Map) to obtain a new forest fire spread simulation.

HLA In 1996 and spurred by the U.S. Army, the High Level Architecture (HLA [Dahmann et al., 1997]) became a standard that defines how to create a global simulation composed of distributed simulations interacting without being recoded. In HLA, each simulation is called federated, and it interacts with other federated simulations in what is named in HLA a federation, which is actually a group of federated. In HLA, communication is established by sharing, dissemination and reception of information. Communication between federated is managed by a Run-Time Infrastructure (RTI). A High Level Architecture consists of the following components: Interface Specification, that defines how HLA compliant simulators interact with the Run-Time Infrastructure (RTI). The RTI provides a programming library and an application programming interface (API) compliant to the interface specification; Object Model Template (OMT), that specifies what information is communicated between simulations, and how it is documented; Rules, that simulations must obey in order to be compliant to the standard.

Open Simulation Architecture OSA (Open Simulation Architecture) [Dalle, 2006, Dalle, 2007a] is the prototyping software platform used in this thesis to experiment and validate our new concepts.

OSA is a collaborative *platform* for component-based discrete-event simulation. It has been created to support both M&S studies and research on M&S techniques and methodology. The OSA project started from the observation that despite no single simulation software seems to be perfect, most of the elements required to make a perfect simulator already exist as part of existing simulators. Hence, the particular area of research that motivated the OSA project is to investigate practical means of reusing and combining any valuable piece of M&S software at large, including models, simulation engines and algorithms, and supporting tools for the M&S methodology.

OSA has been designed as a *layered architecture* in which each layer is devoted to a particular M&S activity or concern, e.g. development, systems modeling, simulation, execution control, deployment, platform administration, and testing. Each layer is designed to be self-contained while still offering the possibility to overload existing layers. As each layer describes a set of components that can be extended or modified by other layers, it makes reuse easier. Indeed, when reusing existing components, a common issue is that some adaptations are usually required in order to match the requirements of the new usage context. In the case of OSA, these adaptations can be limited to the reusing context without requiring changes on the original implementation (which might be used in other contexts).

This layered architecture is inherited from the FRACTAL component framework [Bruneton et al., 2006], which is the basis of our component-based architecture. FRACTAL is a hierarchical component framework offering some benefits such as shared components [Dalle et al., 2008]. FRACTAL comes with an Architecture Description Language (ADL) called FRACTAL ADL, based on XML, that fully supports the layering principle described above, by means of advanced object oriented constructions such as heritage and overloading of ADL definitions. Another interesting feature of FRACTAL is the fact that it supports the addition of any number of non-functional concerns, by means of dedicated controllers, placed in the membrane of the components. Common examples of such concerns are persistence, distributed execution, life-cycle management, naming, and binding. In OSA, such specific controllers are used for the particular needs of M&S (simulation life-cycle, instrumentation, event scheduling, and so on). FRACTAL and FRACTAL ADL are further described in Section 2.4.

OSA is also meant to become a *front-end / back-end* architecture, with plans to rely on Eclipse as a front-end graphical user interface for the definition of the various inputs needed in a simulation experiment. However, the necessary Eclipse plug-ins are still under development or still need to be integrated when reused from other simulators (e.g. the statistical analysis tools from the Omnet++ simulator [Varga, 2001]).

At the begining of this thesis, OSA included a simple simulation engine called SPR (Single Process with Reentrance) based on the AOKell Fractal Component implementation[Seinturier et al., 2005]; this engine implementation could barely be distributed for parallel execution despite tools such as FractalRMI would allow it, because

of the centralized design of the simulation engine and some limitations of the FractalRMI factory that would limit the maximum size of a simulation to a few thousands of components; the compilation of an OSA project was performed locally, using the `ant` Java building tool; although OSA already included an extension of FractalADL for defining observation probes, it was missing the processing part of the instrumentation framework.

As we will see throughout the remaining of this thesis, our contributions to OSA include a redesigned engine based on the Fractal/Koch membrane compiler, an optimized distributed execution mode that allows OSA to scale to configurations of up to millions of components, a network-centric architecture based on `maven`, which also proviodes a new multi-layer project layout, and a full-featured instrumentation framework.

2.3.3 Reusing Techniques in Modeling and Simulation

We differentiate through simulators presented above 6 different ways to reuse:

- Model reuse: NS2/3 and Omnet++ allow to reuse model in several studies thanks to the NED or Otcl languages. Models can also be reused in bigger models thanks to component-based approach.

- Simulation reuse: HLA lets you assemble and operate all the different simulations that could not communicate directly. In HLA, reuse and interoperation are limited within the federation.

- Formal reuse: DEVS offers syntactical reuse through a universal specification independent of implementation. A formal model will behave the same way regardless of the implementation adopting the DEVS specification.

- Software reuse: JAMES II proposes reuse at all levels of the simulation through a plugin architecture. Reuse of different plugins and their assemblies provides a simulation perfectly adapted to the needs of the user.

- Mashup reuse: CD++ proposes to combine application and data using web services within the simulation. Integration of real life information can be very interesting in simulation to predict for example the evolution of a system in case of emergency (fire forest, hurricane, ...).

- System reuse: Sometimes real elements of the system under study can be reused in the simulation, a technique which is also called emulation. For example, this technique is often used in network simulation, where the network stacks implementations within the Operating System kernel can be reused withing simulations. In this case, for example, one solution is to change the OS interface library using LD_PRELOAD on unix systems. Another solution (found in NS3 [Lacage, 2010]) is to use an ELF dynamic loader to run the same application binary into the simulation or into the system.

These techniques are not antagonistic and can be used together.

2.3.4 Open Questions for Reuse

Instrumentation Although some authors carefully describe the implementation details of a simulator and classical discrete-event simulation algorithms (e.g. Banks et al. in [Banks et al., 2004, Andradóttir, 1998], or Fujimoto in [Fujimoto, 2000]), 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 [Andradóttir, 1998], propose techniques to reduce the computational complexity of this dynamic observation and on-line statistics computation.

Others, like [Himmelspach et al., 2008], 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 *instrumenters* 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 JAMES II, the model must notify observers that variables have changed. This prevents reuse without source modification of model that was not originally made for JAMES II.

In [Zeigler, 1984], 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 [Gulyas and Kozsik, 1999], Gulyas and Kozsik address the issue of separation of concerns in simulation using AOP. But their application of the AOP paradigm is limited to the gathering of simulation data. They do not consider using AOP for instrumentation and analysis.

In [Varga and Hornig, 2008], 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.

Integration In the previous section we discussed the benefits of reuse, but often it is confined to the reuse within a single simulation. For example, simulation frameworks offer capabilities to facilitate the production of generic models. In some situations, it may prove to be valuable to reuse elements coming from other simulators. Such elements include reference models as well as engines, simulation tools, or even part of real systems running in real time. It is also interesting to integrate a simulator as a back-end, particularly to

reuse pre-processing tools such as experimental planning.

In the simulation platforms presented above, only HLA-compliant simulators and CD++ have developed techniques for integration. Integration within HLA has a cost since it is necessary that simulators develop a communication layer compatible with HLA. In addition, the integration is confined to the reuse of entire simulation within federations. Mashup technique used in CD++ allow the integration of any software or data through web services. It's really interesting to integrate content from the world wild web but ineffective for the integration of simulation tools that do not offer web services. Emulation is another technique for software integration. An emulator is a program that allows to run software on a platform which it is not intended for. The simulator emulates the underlying system expected by the software. This implies that the simulator intercepts the system calls of the software and returns responses to the software which runs on the targeted system. NS3 offers an emulation solution for the integration of simulation in real networks or the integration of real network node in the simulation. Another solution come from software engineering: the encapsulation in components. The component-oriented programming allows a software unit to be assembled with other components through encapsulation in a component. Integration without modification in components can be done in different ways: using the inheritance feature offered by object-oriented programming, or using aspect-oriented programming.

2.4 Software and Reusing Techniques Used In This Thesis

This section presents software tools and techniques used in this thesis. Section 2.4.1 presents the Fractal component model. Then, section 2.4.2 presents the architecture description language associated with the Fractal Component model. Section 2.4.3 presents the aspect-oriented programming paradigm. Finally, section 2.4.4 presents the Apache Maven project management and comprehension tool.

2.4.1 FRACTAL

FRACTAL is the ObjectWeb Consortium component reference model [Bruneton et al., 2004]. FRACTAL is neither a software environment nor a runtime 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.

The FRACTAL specification defines several levels and sublevels of compliance. These levels allow an implementation not willing or not able to implement completely the model to state how much of the specification it complies with. At the lowest level, a component architecture claiming to be compliant with level 0.0 is just supposed to implement its components using the object programming paradigm. At the highest level, a component architecture claiming to be compliant with level 3.3 is supposed to fully implement all the

features of the specification.

Compared to standard (Java) object instances, components have the ability to support (or obey to) non-functional concerns, such as life-cycle, naming, access control or persistence to name a few. More precisely, "non-functional" means that it is a concern that is not related to the business logic of a given component, but applies equally to all components. Let's look closer at an example with the life-cycle concern: the life-cycle concern is about starting and stopping a component without compromising the whole application; it is meant for high-availability applications, to allow any component to be safely replaced with an upgraded version without shutting down the whole application. This concern applies equally to all the components of the application regardless of their specific business: it is a non-functional concern.

In FRACTAL, these non-functional concerns are implemented by means of dedicated controllers, placed beside the functional code (or merged with it, depending on the FRACTAL implementation). The list of controllers associated to a given component is merged into an entity called a *membrane* in the FRACTAL jargon. Compared to other component models, an interesting feature of FRACTAL is that it allows to build new custom membranes, by adding, removing or replacing any such controller to or from an existing membrane.

Despite most of the FRACTAL implementations come with a minimal set of default controllers, none are explicitly required by the FRACTAL specification. This lack of minimal requirement makes the component model extremely versatile, but it incurs a slightly heavier programming cost, because the exact list of available controllers must be retrieved by introspection. For example, let's consider the so-called `naming-controller`, which is in charge of assigning a name (any string value) to a component. Because this very basic feature is optional, the corresponding controller is not required to be present. Therefore, a careful programming requires that introspection is used to retrieve that controller prior to using it, because there is a risk that it might not be available. This programming constraint is a deliberate choice of the FCM designers. On one hand, if that naming feature was required to be present in all components, then the programming would be easier because the corresponding controller could be accessed without caution. On the other hand, forcing any feature that could be useless to be associated to all component might result in very big components each possibly having a significant amount of useless code (and bugs).

Hereafter, we summarize some of these key features (see [Bruneton et al., 2004] 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 identified by a string name. 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.

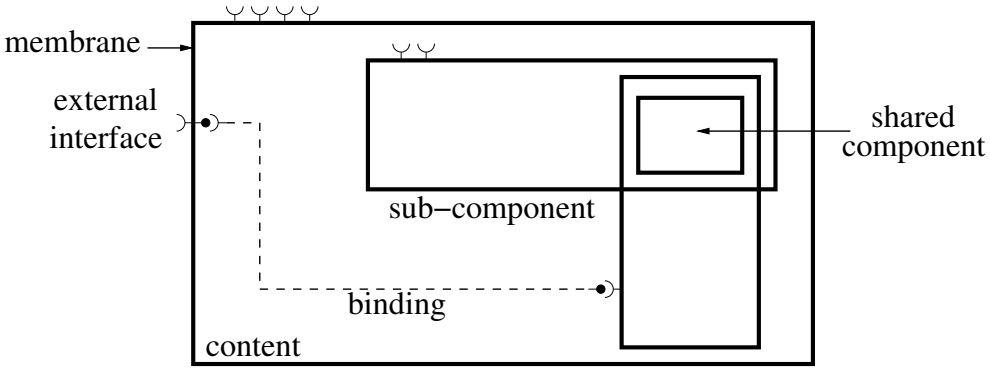

Figure 2.1: FRACTAL component model.

Membranes The control logic is defined in the FRACTAL component model with a membrane composed of controllers, each one being specialized with a particular control mechanism (binding management, lifecycle, etc.). In [Seinturier et al., 2006], Seinturier et al. apply to the design of the control layer the same principles which were applied to the application layer: engineer the control with components. By "contractually specifying the interfaces" [Szyperski, 2002] of these control components, they foster their reuse, clarify the architecture of the membrane, and ease the development of new ones.

The most widely used control membrane in FRACTAL applications is the one associated with primitive components. The architecture of this membrane is illustrated in figure 2.2. This membrane provides five controllers, for managing the lifecycle (`LC`), the bindings (`BC`), the component name (`NC`), the super components (`SC`) and a controller (`Comp`) implementing the general `Component` interface, which is available for all FRACTAL components. As a matter of convention, provided interfaces are drawn on the left side of the components, and required interfaces are on their right side. Bindings represent communication paths between the controllers.

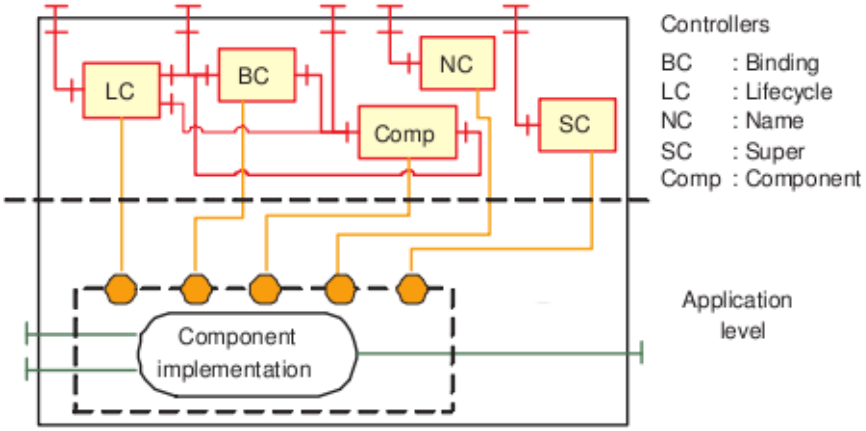

Figure 2.2: Primitive membrane: control level for primitive components.

The architecture presented in figure 2.2 illustrates that the control function for primitive components is not simply realized by five isolated controllers, but is the result of the collaboration of these five controllers. Compared to a purely object-oriented approach, a component-based solution for the implementation of control membranes allows describing explicitly the dependencies between controllers. New control membranes can be developed by extending existing ones, or simply by developing a whole new architecture.

From a simulation point of view, easily develop controllers allow to quickly implement new formalisms, new algorithms, or add non-functional services such as persistence to simulation components.

Hierarchical structure Components may have a hierarchical structure (fig 2.1). 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 membrane 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 membrane or inside its neighbouring (inner part) components.

Interface 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.

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. It is offered to 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 to appear in the content of several distinct enclosing components. Such components are called shared components. This property has two noticeable consequences: (i) a component is possibly 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.

2.4.2 FRACTAL ADL

The FRACTAL Architecture Description Language (FRACTAL ADL) is a contributed software Library, written in Java, which is part of the ObjectWeb Consortium's FRACTAL project. FRACTAL ADL provides a Factory component that reads architectures descriptions from files and build the corresponding hierarchical component-based software architecture in memory. These architecture descriptions are provided as XML definitions, according to a Document Type Definition (DTD).

The FRACTAL ADL Library is built using a collection of FRACTAL components. Interestingly the component assembly that forms the FRACTAL ADL factory component is built recursively: it reads its own architecture description (i.e., the architecture of the hierarchical components used to implement the FRACTAL ADL factory) using a hard-coded bootstrap component architecture. Thanks to this flexible, reflexive architecture, the FRACTAL ADL components can be extended at will, which in turn allows to extend the ADL itself, and therefore the language definitions it is able to recognize. This flexibility might seem excessive, but it is consistent with the FRACTAL philosophy described earlier, in which the non-functional services provided by the membrane of a component can be customized and extended at will. This ability has been used to extend the original ADL in various directions, such as including support for the distributed execution of components for example. In the OSA project[Dalle, 2007b], we used this extension capability to allow the scheduling of exogenous events directly within a (model) architecture definition, or to specify the points in the modeling code where to collect data samples for the instrumentation framework.

Although almost all the content of the Factory could be re-engineered, and therefore almost all its functional specifications could be changed, a typical FRACTAL ADL Factory supports the following constructs :

- definition of a component, which is a container for more definitions, specifying its name and source (either binary code or another ADL definition file),
- specification of component interfaces (services offered and used),
- list of components bindings (how services offered by some components are connected to services used by others)
- component location (on which host to deploy the component instance for execution)
- component content (list of sub-components in case of a hierarchical component)

Listing 2.1: A sample FRACTAL ADL declaration that defines an application made of client and a server.

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

<definition name="ClientServeurApp">
  <component name="Client">
    <interface name="cli" role="client" signature="ClientSvc"/>
    <content class="ClientImpl"/>
  </component>

  <component name="Server">
    <interface name="srv" role="server" signature="ServerSvc"/>
    <content class="ServerImpl"/>
  </component>

  <binding client="Client.cli" server="Server.srv"/>
</definition>
```

- component special features (e.g. the template feature described later on, or the capability of scheduling of simulation events used in OSA)

The Listing 2.1 illustrates the previous basic constructs through a simple client-server example: at the top level, the application is composed of two components, a client, named "client" and a server named "Server". Since the semantics of each XML tag-word is self-explanatory, it is not to be further explained.

2.4.3 Aspect-Oriented Programming

AOP [Kiczales et al., 1997] 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, which leads to the "hyper-spaghetti" phenomenon. 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.

Several languages and frameworks are available for programming aspect-oriented applications such as AspectJ [Kiczales et al., 2001] for the java programming languages or AspectC++[Spinczyk et al., 2002] for the C++ programming languages.

2.4.4 Maven

Apache Maven project is a project management and comprehension tool, developed by The Apache Software Foundation. Apache Maven is a free software tool for managing and automating production of Java software. The goal is comparable to Make under Unix: produces a software from source, optimizes the tasks necessary for this purpose and ensures the production order.

Maven uses a paradigm known as the Project Object Model (POM) to describe a software project. Each project or subproject is configured by a POM that contains the information necessary to process the Maven project (project name, version number, dependencies to other projects, libraries needed to compile, names of contributors, and so on). This POM is materialized by a pom.xml file at the root of the project. This approach allows the inheritance of properties of the parent project. If a property is overridden in the POM project, it covers those who are defined in the parent project. This allows to reuse configuration.

Another important feature and relatively specific to Maven is its dependency management. Indeed, Maven can automatically download the dependencies of a project. Maven can also publish a project on a repository in order to make it available to other projects. The Maven dependency management is simplified by the notions of inheritance and transitivity. Maven allows the use of different categories of repositories:

- **local repository** includes everything that the developer has used and developed.
- **global repository** includes everything that have been publish publicly by the developers.
- **enterprise repository** makes available to all developers of a company private projects.

The use of these repositories allows versioning, and thus promotes replayability by tracing and downloading all the dependencies for a given version of a project.

Maven provides a plugin architecture for adding new features. These plugins are available on Maven repositories, or can be developed. Thus, Maven can be configured to suit our need.

2.5 Discussion

Recent success stories in Software Development and Designs have put the light on new elements of discussion worth to consider in order to make reuse a true daily reality.

One of these is the Eclipse success story [des Rivières and Wiegand, 2004]. The success of the Eclipse development platform is mainly due to a simple but rather revolutionary

philosophy: let everyone plug and "play" with what they want in the software platform. This is a philosophy more than just a technical solution, because the technical solution (a plug-in-based architecture) comes with strong incentives for reuse. The Eclipse core is only a minimal environment providing little functionality if we except its highly versatile Graphical User Interface and its powerful plug-in management system based on OSGi bundles [Gruber et al., 2005]. Indeed, in terms of ergonomy, it is very difficult to find the perfect design that will please every end-user. Hence, the more a complex software includes a large set of default functionalities, the more it has chances to distress its potential end-users.

Going one step further in the analysis, it appears that because Eclipse has such an ability to adapt to the users needs, and in particular not to force them to stick to a particular solution, and because early Eclipse contributions where plug-ins to support Eclipse plug-ins development themselves, it adequately and conveniently supports any specific corporate culture (provided that new plug-ins are developed to support this culture). Indeed, many software companies have their own legacy corporate methods and procedures. Therefore, the decision to move from old but well known developments tools supporting these legacy methods and procedures to new tools is often perceived as a major risk. The ability of Eclipse to embed such legacy methods and corporate procedures by means of dedicated plug-ins is certainly another key of its success in the software industry.

However, the challenge with such a philosophy is to *prime the pump* of contributions and initiate a virtuous circle: as long as no plug-in is available, nobody wants to use an empty shell, and therefore nobody is interested in developing new plug-ins for such a platform. In order to make such a product appealing, a minimal set of functionalities needs to be provided by the software authors, in addition to the core functionalities. This is where Eclipse made a difference: instead of providing these functionalities as a fixed immutable set, they are provided as regular, and thus *replaceable* plug-ins.

Chapter 3

Design Considerations

Contents

In this chapter[1], we consider design questions for M&S software. We claim that discussing, designing, developing, and comparing M&S products should start with software engineering considerations. We shortly introduce some of these engineering concepts and discuss how these relate to the M&S domain.

All processes in M&S can be seen as software engineering, and thus we discuss herein from an engineering point of view which engineering techniques we could apply to improve the overall quality. M&S "engineering" may be considered according to three main complementary dimensions: either we consider the design of a M&S software product or we consider the design of a model or we consider the design of a simulation (i.e. experiment design). In the following we focus on the first term which is essentially software engineering, because we strongly believe that a good software design is the first requirement for achieving trustworthy results as soon as a computer is used to produce these results. Nevertheless many of the techniques hereby discussed can be used for all M&S engineering dimensions.

The question addressed in this chapter is to identify which requirements a good M&S software product must fulfill, and how software engineering can help in fulfilling these requirements. Obviously, we must consider the software dimension of such products, and refer to the abundant literature about software engineering. For example, Sommerville states that a good software should: "... deliver the required functionality and performance to the user and [*it*] should be maintainable, dependable and usable." [Sommerville, 2007, p. 6].

Hence, we can first use software engineering to understand M&S tool building as a well-defined project, with feature and time management, using a development paradigm (a process model to be used, e.g. V-Model, waterfall; a programming habit, e.g. eXtreme Programming), and so on. In addition we can apply software engineering methods to design the M&S software product, e.g. pattern based design description [Gamma et al., 1995], we can and should apply testing techniques to our software product, we can define maintenance rules for the software, we can try to find (architectural) patterns which may help on creating M&S tools [Garlan and Perry, 1995], and last but not least we can control the overall production process (e.g. according to ISO 9001). Experience shows that a well-defined development process as well as reuse can increase the quality ([Endres and Rombach, 2003, pp. 77], [Sommerville, 2007, p. 417]). Well-designed M&S software products should support reuse of existing (software) components. Through reuse, such components can be continuously improved in time (both in terms of quality/reliability and efficiency), and gain a better understanding from their (re)users community. Therefore, in the end, reuse is expected to make solutions more mature and ultimately improve the efficiency of our research efforts.

In the following we sketch different techniques and patterns which can be used to discuss, to design, to develop, and to compare M&S software in general.

[1] This chapter was written in collaboration with Jan Himmelspach and Olivier Dalle and has been published in the 41st Winter Simulation Conference (WSC) in Austin, Texas [Dalle et al., 2010].

3.1 Engineering in M&S

Engineering techniques can be applied in modeling, experiment design, and M&S software product design and development. The very first engineering discipline to be taken into account is essentially software engineering – the result is a software product with which we can start to apply M&S to applications, or with which we can start to do M&S research. Software engineering methods generally aim at the production of high-quality software in a cost effective way.

Having a M&S product to work with we can start with "model engineering" – i.e. we start a model building process including calibration and validation steps. Having both a model and a M&S software we can execute our experiments with, we can finally start with the engineering process of our experiments. For all three engineering tasks, common engineering techniques can and should be used.

In the following we give an overview of relevant software engineering techniques to be used if a M&S software is being developed.

3.1.1 Process and Development Models

Process models play an important role in the definition of "projects". They make clear what has to be part of the project, and when we have to expect to enter a new phase and when we might have to enter an old phase again.

There are some specialized process models for M&S around as well [Law, 2007, Balci, 2003, Sargent, 2008, Van Waveren et al., 2000, Wang and Lehmann, 2008]. These adapted models define and align different phases in a M&S project, however they all lack the (explicit) phase of the software development. Partially this might be related to the fact that often no precise distinction is made between model, simulation algorithm, and simulation, nor between a general and a domain specific solution, and that often the idea of "reuse of structures" is not considered at all.

3.1.1.1 Process models

The increasing number of process models can be applied whenever something is about to be produced / constructed. Among these the waterfall model, v-model, sequential model, concurrent model, prototyping (vertical, horizontal), evolutionary model, component based model, and spiral model. All these process models have in common that they define how the development process will flow – thus do we have to expect cycles, how many main phases do we have, and what are the phases? These models thereby help to manage the project – often a phase can only be entered if the previous phase has been finished, often phases are associated with milestones – and thus you definitely know whether you have finished a phase or not, and they label the phases, and thus define what has to be done for a successful progress.

The process models from the M&S domain focus on the processing of simulation studies. Thus they typically encapsulate the path from model building over experiment design to experiment execution. Typically the recommendation is that you explicitly select a pro-

cess model for all types of engineering. If you consider the complete development process comprising software engineering, model engineering, and simulation engineering then you might end up using different process models per task (or a M&S specific one for the latter two) and another process model for the overall (here 3/2-step) process.

3.1.1.2 Development models

Another important aspect on developing software is the organization of the development. Various team structures can be used (hierarchical structures (e.g. chief programmer) versus flat structures (e.g. egoless programming)), team members can work on their own, or they work together (e.g. in eXtreme Programming (XP)). However, not all organizational forms can be used for all team settings. In the academic world teams are often pretty small, sometimes they comprise exactly one member, e.g. someone working on her/his thesis, and thus some development models cannot be applied at all (e.g. pair programming in XP).

3.1.2 Team Management

The composition of a team (e.g. you could try to get complementary personalities of the team members), the skills of the team members, their motivation as well as the work setting have a major impact on the overall development process. Consequently a good team management is one of the key factors for a successful project. Any project management is essentially based on a good team management: you will not be able to meet your project goals with too few, unexperienced, unmotivated or overstrained people. This gets of special importance in the academic setting where team members might be undergraduates or just graduated ones (low(er) experience level) and it might be especially bad in the field of M&S due to the number of different tasks to be done for M&S software and the knowledge required to do so.

3.1.3 Project Management

To understand the work on a given subject as project, including time and resource management, is considered to be a corner stone of a successful project. The management of a project has to be aligned with the process and the development model to be used. If all potential engineering tasks (tools, model, and simulation development) shall be applied within one large M&S project, project management gets even more importance – we need a good time management with milestones, which precisely define when we can start with subsequent engineering tasks. Project management should include quality management. I.e. we have to take quality concerns seriously, and we have to integrate steps which try to check whether we have reached the quality we are interested in.

3.1.4 Quality Management

Quality is a multifaceted, but widely used term [Himmelspach and Uhrmacher, 2009b]. And although many different notions of quality exist, almost everyone agrees to the state-

ment that "quality" is of importance. Means to improve "quality" are around for quite a while now, and they are intensively applied. A relatively well-known example is the ISO 9001, and its application in many different industries. Some of the notions of quality focus on the production as such – they try to define minimal aspects to be taken into account, e.g. they might request a minimal documentation of the process applied. In research this was / and still is done by the publication process. However, sadly, the latter is often of poor quality [Pawlikowski et al., 2002], and thus standards used in other domains might get of increasing interest here as well. Typical means to increase the quality are validation, verification, and well-defined development rules (as coding styles, repository usage, ...).

3.1.5 Design and Documentation of Products

At latest from 1995 on [Gamma et al., 1995] the usage of design patterns to describe parts of software started to be common. Together with architectural patterns they can be used to describe a system without the need to name all implementation details from the beginning on, but still being relatively precise.

To use such descriptions can help to direct those who are going to realize a concrete piece of software. Software pieces developed should fit into the overall architecture, and therefore the architecture has to be well-defined, because the variety of alternatives on the realization of a concrete piece of functionality is large. The latter includes alternatives on the implementation / algorithmic level (e.g. different event queues, random number generators, simulation algorithms), as well as interfaces, and the overall design concept (e.g. are these "pieces" services, plug-ins, are they to be included in a server, are they in the back- or front-end, what shall be reusable). In addition a good documentation is essential for *reuse*. To use abstract descriptions can help to make the product reusable at all because then people do not need to understand all implementation details until they can make a first judgment about the re-usability. And it is a good base for discussions about and the comparison of products. In the following we mostly focus on design considerations under the aspect of "reuse".

3.1.6 Engineering Requirements

Sophisticated engineering requires knowledge of its methods and tools. This comprises here knowledge in software engineering, knowledge in experimental design, and knowledge in modeling. To execute all required engineering steps thoroughly requires a lot of time. But time is a limited resource, as most often man power is as well. Thus the main question is: where can we save time? If the first step of the overall engineering process is software development we should take a closer look at this one – especially if all subsequent steps are based on this. This gets even more important if we take a look at the ever growing number of M&S products – why do we always have to redo everything from scratch?

3.2 Modeling and Simulation Application Design Considerations

If we start to develop a new high-quality M&S software product we have to get clear about a variety of issues. At first we need to make a number of top level decisions which are often hard to revise later on:

- intended use of the product (modeling, simulation, method development, reuse, ...)
- intended user group
- general architecture of the software
- functionality to be included

In addition, due to the widespreaded usage of M&S it seems to be recommendable to setup a glossary of commonly used terms. This can help to avoid misunderstandings stemming from commonly used terms with different notions as model, simulation, and experiment. These considerations might provide a base for the initial design decision: the general architecture of a the M&S software product to be developed. In addition, due to the widespread usage of M&S, the requirements for a "good" M&S software may differ – because different users might have a different notion of quality [Himmelspach and Uhrmacher, 2009b]. Independent from any decision, the major requirement for M&S is credibility, i.e. results of M&S should be reliable. Therefore it is essential that developers get aware of the overall number, type, and interactions between the "bricks" of the software to be created. For M&S this means that we we need to decide which "desirable software features" [Law, 2007, page 193 ff] we'd like to include from the overall set of possible techniques and elements [Himmelspach, 2009]. Keys to fulfill the major requirement of credibility are a careful development of the overall product and of all elements. Therefore a careful VV&A process of everything developed is mandatory, in particular this is well-known for modeling, and it is something which should be supported by M&S software in general [Balci and Nance, 1992]. But we need the same for the M&S products the model is created in / with.

3.2.1 Software Design

Architectural designs and design patterns are important aids if an application shall be described. They help to think about problems and solutions in a more focused, abstract, and nearly standardized manner. Using explicit architectural designs and design patterns can thus improve the overall development process. They can help on discussing problems, and they can lead developers to reusable solutions fitting to the overall system being developed [Sommerville, 2007, p. 293].

In the following we take a look at a variety of architectural design alternatives for applications, and we try to map them on M&S software: Are they usable at all, and if so, are they usable for complete M&S products or just for parts of it?

3.2.1.1 Architectural design alternatives

The list of design alternatives given here is not complete and should thus be considered as a "teaser" to motivate a search for additional ones if none of the alternatives listed here is the right one for your purpose.

Model view controller A Model-view-controller (MVC) is a pattern for an architecture which separates the model (data) and the controller (control logics), from any number of views on the data. In M&S we can exploit this pattern for different tasks such as modeling and simulation execution. For modeling, the MVC pattern can be exploited to have different concurrent views on the model to be created. In simulation, the MVC pattern can be used to describe the dependency between an (interactive) runtime visualization, the executable model, and the simulation algorithm.

Layers A layers based architecture is an architecture in which high-level components depend on low-level components which further depend on even lower-level components and so on. The layered architecture supports modifiability, portability and reusability of each layer independently from the others thanks to the vertical decomposition.

In M&S layered approaches can be used to describe solutions for model composition, model instrumentation, simulation execution, and the interplay of these.

Blackboard A blackboard based architecture is based on a centralized information exchange space – the "blackboard". This architecture has been used in artificial intelligence (AI) products, for example. Information is written to the blackboard and all involved entities can read the information they are interested in. In AI this has been used to realize cooperative problem solving strategies. In M&S blackboard based approaches can be used for data collection, and for synchronizing simulation algorithms.

Client-server A client-server based architecture is an architecture for distributed applications, including distributed M&S products. These can be peer-to-peer, 2-tier, n-tier or Cloud/Grid Computing products. This architecture can be used in two ways: a server knows about the clients and can delegate jobs to these, or the clients send jobs to the server. In M&S client-server architectures can be used for distributed computation, for data collection, for model databases, and for data analysis.

Front-end and back-end Front-end and back-end architectures separate the overall process of the application into two phases: in the front-end data is collected which is then used by the back-end to perform operations on. Thus the front-end can be considered to be the interface between users and the back-end. A strict distinction between front-end and back-end can be found in M&S products as well – if models or experiments are designed they are typically designed in the front-end and need to be transformed for the back-end. This is used if models are created in a special modeling language, and if they are transferred to a representation which can be executed in an efficient manner.

Monolithic application A monolithic application takes care of everything on its own. Usually parts of such an application cannot be reused, and they cannot be easily exchanged. M&S tools might be created in this manner. If so, they are often created to compute a concrete particular simulation (with one model), on a single platform.

Service-oriented architecture A Service Oriented Architecture (SOA) provides the systems functionality by a set of inter operating services. The services are only loosely coupled, and systems based on this concept are especially suited for distributed computing scenarios. Each service provides a well-defined function. A service does not depend on the context or state of other services. Service-orientation can be used for M&S software as well. Either services are just from the M&S software to realize a certain functionality (e.g. databases or visualizations) or they can be realized as fully service-oriented architectures (e.g. simulation algorithm, modeling front-end, and random number generators as services).

Pipes and filters Pipes and filters (also known as pipelining) depicts a system comprising independent functional units each working on an input which is transformed by the unit into an output. These functional units are combined in a chain, which means that the output of the predecessor is the input of the successor. In M&S this architecture can for example be used to realize an automated experiment execution and post processing of the simulation data.

Plug-in architecture A plug-in (also known as add-in, add-on, snap-in, or extension) based architecture allows to extend an existing application with new functionality without the need to recompile the application. This functionality might be provided by third parties, and thus it allows to integrate unforeseen features, helps to keep the application small (you only need to include what you are in need of), and it helps on integrating software distributed with different licenses. Plug-in architectures can be built on existing bases like the Java Plug-in Framework or OSGi. M&S software can be created based on a plug-in concept as well. Thereby plug-ins can be exploited on a variety of levels, pursuing a strict separation of concerns and making reuse possible and in our opinion "relatively easy". Plug-ins in a M&S software can be, for example, simulation algorithms, modeling languages, optimization algorithms, event queues, random number generators.

Mixed architectures Sometimes several architectures are mixed for the creation of a particular application. Mixing architectures is of interest if none of the standalone architectures can be used to describe the overall architecture of the software to be developed. In M&S this seems to be common approach: M&S software can contain relatively independent parts to support modeling, simulation run execution, and experiment definition and control. Each of these might be realized using a different architecture, e.g. simulation execution might be realized based on a "client-server" architecture, whereby the the modeling might be based on a "front-end – back-end" system.

3.2.1.2 About "classes" of M&S products

Many M&S products are labeled with one of the terms "library", "framework", "kernel", "platform", "tool", "workbench" or "environment". It is hard to decide whether the name is chosen correctly if the tools are not described in a sufficient manner, and/or if the code is not fully available. Usually the name should indicate the type of the product, and thus give a first hint on how one can use it. Consequently we should take care of using these. Libraries are collections of reusable functionality. Frameworks provide in addition to reusable functions – as libraries already do – "flows of control" [Johnson and Foote, 1988]. A framework may be built on top of a set of libraries, and a framework might be used to create more specialized solutions. Frameworks shall ease / speed up the development of software from the domain they are created for, and they usually can [Madsen, 2003]. A kernel typically is the lowest software level available and comprises data and process management. It is the base all software parts runnable on this kernel have to be built on. A middleware typically provides support for the integration of components (e.g. CORBA), this might include inter component communication, security, resource allocation, and transaction management. A tool usually provides support for an individual task (e.g. a compiler, word processor). They can be distributed as general-purpose, standalone tools or they might be integrated into a workbench. A workbench is a set of tools to support different process phases of the production process. An environment typically supports all of or at least a substantial part of the production process it has been created for. Therefore it might integrate a number of workbenches [Sommerville, 2007, p. 87]. A platform typically means a background system which provides the basics for other products to run on (e.g. Java VM).

3.2.2 Open Architectures

The benefits of well-defined architectures are manifold and already mentioned above. Here we would like to strengthen the idea of open architectures which can help to create concrete software products effectively. For large companies it might work to have an own architecture, however for smaller groups it is recommended to use existing open architectures [Endres and Rombach, 2003, p. 56]. This helps sharing results, and thus helps on creating credible results in the end – if experts from different domains add their knowledge. But open architectures mean that there is an additional (small) burden for developers using open architectures: they may run into the need to adapt their code to changes in the architectures. However, we think that this burden is less important than the burden to create a credible M&S application from scratch. An open architecture is not bound to any architectural design – every architectural design can be created as an open architecture. Open architectures can be developed from the beginning on as "open architecture" or they might be released after some time of closed development. Open architectures have to be available, and to be usable by everyone who is interested in.

3.2.3 Discussion

Technical solutions for reuse are rather well-known and often adopted in M&S but this is not enough to achieve wide reuse. The technical solution must come with an open philosophy that gives true incentives for everyone to reuse each other's contributions. Of course, some technical solutions, like the High Level Architecture (HLA), have been widely adopted, but this was more by necessity: HLA is a solution when two (or more) simulators *must* be interconnected. In Eclipse, the reuse event is more opportunistic: reuse often takes place in a more general optimization process, in which end-users tends to ever improve the quality of their working environment.

And we believe this form of opportunistic reuse is a very missing element in the M&S field. Indeed, we claim that every single piece of engineering (not only software) may be worth to reuse and we should go further than reusing parts of models by encapsulation or engines through middlewares / RTIs. However, applying reuse everywhere results in one strong technical requirement (in addition to the usual methodology concerns about validating the reuse context): reusable pieces must be sufficiently separated. One solution for solving this issue is to apply the Separation of Concerns (SoC) Software Engineering principle. Examples of pieces that can be reused independently using SoC techniques include the various levels of modeling, instrumentation, scenarios, experiment plans, deployment maps for distributed execution, documentation templates, unit tests, V&V methods.

Net-centric architectures can help to establish such a broad reuse. For example, repositories help to maintain precise version information and allow to track changes, special databases make reusable elements available (and findable) and thus help on reproducing experimental results, and publicly available and commonly used ontologies can help to classify reusable elements.

Another important issue in reuse is licensing. A thorough discussion of the impacts of certain licenses on reusability is out of our expertise but it is a serious issue with which one should carefully deal. Some licenses might restrict reuse or come with constraints that require careful attention, such as the General Public License (GPL). Consequently, for a wide-spread (re-)usage of products, flexible and open licenses should be used. Component/Plug-in based approaches may help here, because they might ship with different licenses as the product they are to be used in – as long as their licenses are compatible with the product they shall be used in. This can lower the barrier to contribute, because the authors can keep full (technical and legal) control over their contribution, they can contribute to different projects or even sell their product in the end.

Chapter 4

Contributions to Reuse

Contents

In this chapter, we investigate practical means of reusing and combining any valuable piece of M&S software at large, including models, simulation engines and algorithms, and supporting tools for the M&S methodology. Then, we focus on how to provide distributed executions means that require no modification on simulation software as well as models. We also present our solution to develop and reuse models, scenarios and engines using aspect-oriented programming and component models such as the FRACTAL Component Model (FCM) through the OSA architecture.

We saw in section 2.3.3 that reuse can take several forms: model reuse, simulation reuse, formal reuse, software reuse, or service reuse. Then we saw in section 3.2.3 that reuse increases dependability, is less error prone, makes better use of complementary expertises, improves standards compliance, and accelerates development. In this chapter, we propose a solution that allows to go further by separating all the elements of a simulation. This strict separation of concerns allows to add, replace or delete all the elements of a simulation. Component programming allows a separation of concerns by components (which act as black box). We can go further with the use of an architecture description language that allows to view a simulation as a set of layers, each layer reflecting a specific concern. There is no limit on the number of layers that makes up a simulation. In addition, the use of software engineering techniques such as aspect-oriented programming, which can reverse dependencies, allows communication between elements defined in separate layers without breaking the strict separation of concerns.

Figure 4.1 shows a simplifed view of the resulting OSA layered architecture. On the left we have a Maven repository containing components. The central part shows the OSA layered approach with simulation engine, models, scenarios and instrumentations. Every layer can be composed by several sub-layers (such as the representation of the model layer, scenario layer or instrumentation layer). Each layer is independent, communication between elements of distinct layer is done either through the component interfaces (which assumes that a layer need another layer to work: low independence) or because of the AOP (strong independence). The composition of all layers (through the mechanism of inheritance and overloading of FRACTAL ADL) leads to a complete simulation architecture.

Section 4.1 describes our motivations and objectives to promote reuse in modeling and simulation software. Section 4.2 describes the use of FRACTAL ADL and AOP through the design and reuse of models and scenarios. Section 4.3 describes the use of "componentized" membranes and the extensibility of FRACTAL ADL through the implementation of a new simulation engine. Section 4.4 describes the use of FRACTAL RMI and FRACTAL BF to distribute a simulation without modifying the existing code. Section 4.5 describes other means for enforcing reuse, such as our work on FRACTAL template to have a iteration control (loop) in the ADL (4.5.1) or the Maven project management tool that enforce reuse and replayability (4.5.2).

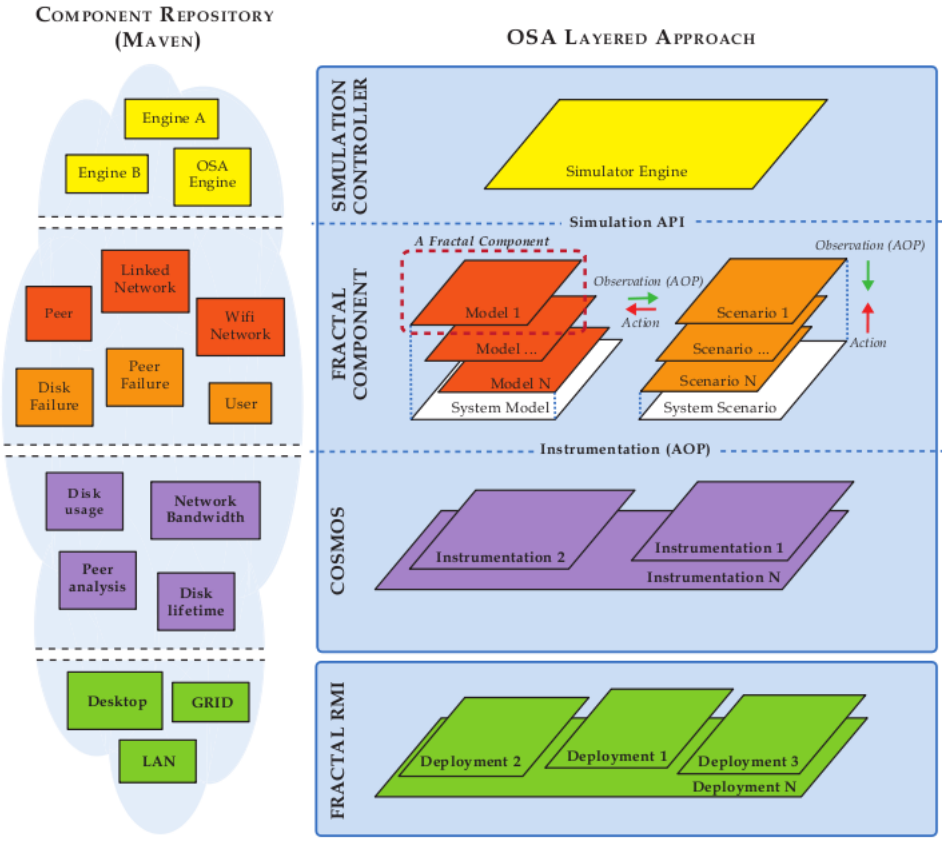

Figure 4.1: A view of the Open Simulation Architecture.

4.1 Motivations and Objectives

By repeating the simulation activities defined in section 2.3.1, we realize that reuse can be beneficial in all the stages of a simulation study. Solutions put forward and listed in section 2.3.3 are not antagonistic and can work together. However, it is imperative to separate concerns or it will be impossible to reuse an existing element of a simulation. Our experimental platform OSA, based on the FRACTAL component model, allows to strictly separate concerns through the use of FRACTAL ADL which allows to see each simulation concern as a separate layer. Following, we list the forms of reuse we consider in this thesis:

- **Reuse of (model) architecture** allows to define an architecture of reference, and have different implementations that conform to this architecture. Thanks to the component-oriented programming and the use of an architecture description language, it is easy to define a reference architecture and simply change the implementation of the components. Reuse of architecture can also prove to be interesting for building to new architectures incrementally, by assembling existing architectures. FRACTAL ADL mechanism provides overload and multiple inheritance to define a

new architecture by the composition of existing architectures. Section 4.5.1 covers the work we have undertaken to add a new mechanism to FRACTAL ADL: iteration control (loop). The iteration control in the ADL allows to define dynamic architectures more easily reusable.

- **Reusing of model** allows a model to be placed in different experimental conditions, using different scenarios. This is the case when we want to test one solution in several situations (test study). **Reusing of scenario** allows a reference scenario to be applied to several models. This is the case when we want to test several solutions in one situation (comparative study). Reuse of model and scenario is studied in detail in section 4.2.

- **Reuse of simulation engine** and the strict separation of concerns between the engine and the rest of the simulation allows to change the implementation of the engine, but also to incorporate new simulation formalisms. This allows reuse of models and engines from other simulators. Section 4.3 details the implementation and the addition of an engine in our experimentation platform OSA. Chapter 6 details the integration of a formalism (DEVS) and an existing implementation (from JAMES II) into our experimentation platform OSA.

- **Reuse of computational resource configurations** allows to test different distributed simulation algorithms for a given configuration of component resources. Reuse is possible through the strict separation of concerns between the model, the architecture and the configuration of the distribution. This reuse is detailed in section 4.4.

- **Reuse of the instrumentation and analysis** allows to apply multiple instrumentations to the same model (this is the case when reusing a model in different studies). It also allows reuse of a given instrumentation on multiple models (this is the case in comparative studies where one wishes to observe the same variables on different but similar models). This form of reuse is studied in detail in chapter 5.

- **Reuse of verification** is interesting for automatically checking different implementations of the same type. The integration of models from other environments allows to use the features of these environments. For example JAMES II provides a platform for automatic verification. It may be interesting to develop models in JAMES II to take advantage of the automatic verification and then use them in another simulation platform such as OSA. We elaborate on this possibility in more details in chapter 4.

- **Reuse of tools** for pre and post processing allows access to reliable and powerful tools. Techniques for reusing tools such as the JAMES II experimental planning are discussed in chapter 4 while techniques for reusing Scave, the post-processing tool of OMNeT++ [Varga, 2001, Varga and Hornig, 2008], are discussed in chapter 5.

4.2 SoC and Reuse in Model and Scenario

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 et al., 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 feedback 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 the following, 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 many time, expenses, and efforts [Davis and Anderson, 2004].

4.2.1 Advanced Scenarios Case Studies

We present hereafter two case studies to illustrate the new techniques we use to build advanced scenarios reusing existing component models. It is worth mentioning that these techniques allow to use 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 4.2 shows 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

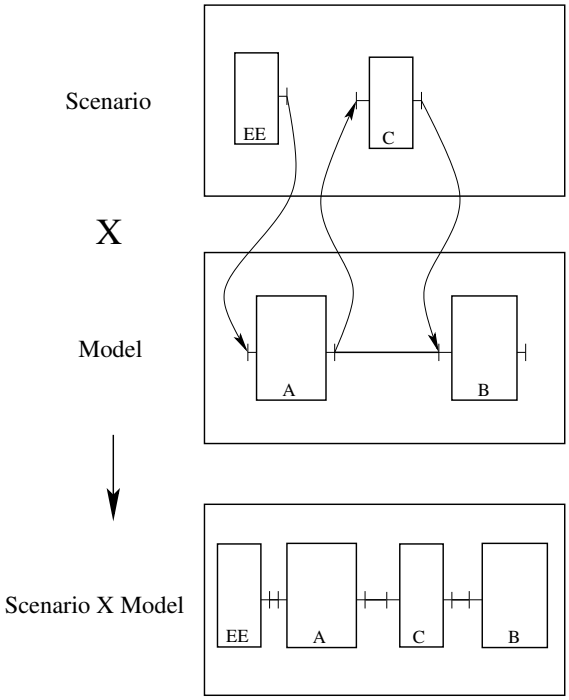

Figure 4.2: Reuse and adapt a model of reference.

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 FRACTAL ADL and (ii) Aspect-Oriented Programming (AOP) like AspectJ. 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.

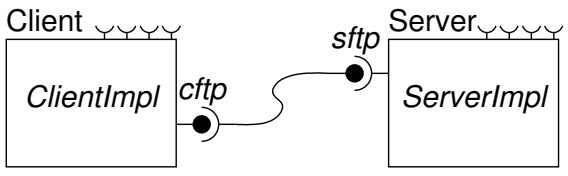

Figure 4.3: Components layout of File Transfers Protocol case study.

As described previously, we use AOP and ADL in an original way to override difficulties in reusing models. We choose 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

Listing 4.1: FRACTAL ADL definition used to implement layout of figure 4.3.

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

Protocol (FTP). This simple model has not been developed in order to be used in this study but we assume it has been successfully validated for the needs of this security study. Also, we assume we are not supposed to have access to the source of that model which is used as a pre-compiled "black-box" model. Figure 4.3 shows the architecture of the model, and listing 4.1 details its implementation in FRACTAL ADL. 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.

The protocol represented by this model is a two-party protocol. We will denote the two parties by the names Client and Server (Client want to be authenticated on Server). The model behavior is as follows : the client sends the user's login and password to the server to be authenticated. To do this, client asks his interface (cftp, declared line 07) to obtain connection with the server. In this study, we focus on the login step 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 FRACTAL ADL. Second, we will show how to simulate a spy-ware on client using the overload capability of FRACTAL ADL and AOP.

4.2.2 Man-in-the-middle Attacker with FRACTAL ADL

From the original model described in section 4.2.1, 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. We assume all

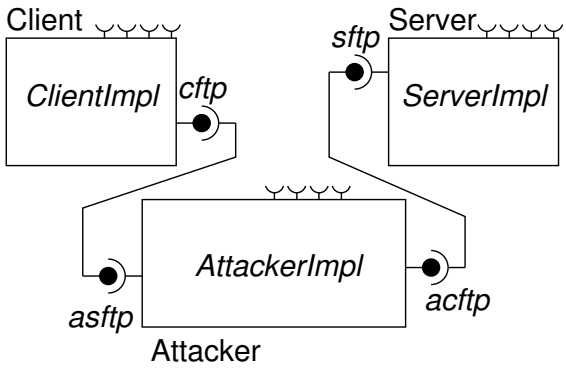

Figure 4.4: Components layout of FRACTAL's MITM attack.

Listing 4.2: FRACTAL ADL definition used to implement layout of figure 4.4.

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

communications between Client and Server are (or might be) intercepted by the Adversary. Thus both Client and Server talk to the Adversary instead of directly to each other. Adversary needs to transmit information between Client and Server, but - it's the security break - he can read, change, or drop messages exchanged between the 2 parties depending on his goals.

What makes this case interesting is the ability to modify the original FTP client-server topology (figure 4.3) to obtain the new topology described in figure 4.4. In practice, to build this scenario we need to add a new component inside an existing model. Like in reality, the Adversary needs to mimic Server's interface and Client's interface. In fact, the Adversary needs to imitates Server for the Client, and imitates Client for the Server. Figure 4.4 shows the new architecture we want to obtain compared to figure 4.3 section 4.2.1. Since the model is locked, we cannot change his topology directly in source code. Listing 4.2 shows how to use the FRACTAL ADL overload capability to

overload the topology. Line 04 shows that we extended the original ftp model in a new model called mitm-ftp. Line 06-14 declares the new Adversary component. And line 16-19 demonstrates how overload the original 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 go through the Adversary.

This example shows how to modify a model to include new component or change topology. The overload capability of FRACTAL ADL permits to reuse and change some specification of the model, like the topology. In fact, in our example, communications 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 FRACTAL ADL to add a new component and change the topology, but we also demonstrate how to use AOP. The next section describes the FTP model with a spy-ware inside the client.

4.2.3 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[1] [Stafford and Urbaczewski,] into the Client from the original FTP model. The goal of this attack is to steal 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 4.5. We see the Client is connected to a third entity (Spy) that implements the Spy-Ware.

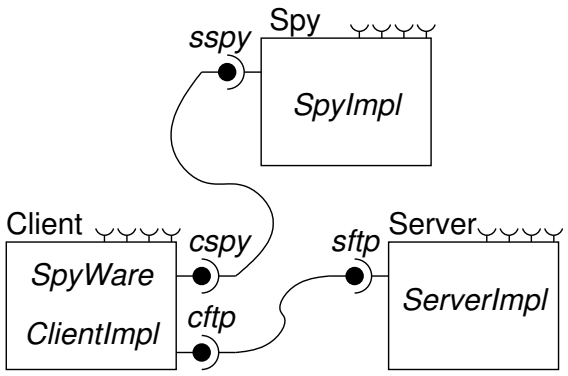

Figure 4.5: FTP model with Spy-Ware in Client.

Listing 4.3 shows a solution using FRACTAL ADL and AOP to introduce spy-ware in the 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 the Spy together. Line 04 shows how to create a new model extending the original FTP model. Lines 06-17 represent the Spy component, lines 07-09 represent

[1] 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].

Listing 4.3: FRACTAL ADL used to implement layout of figure 4.5.

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

the interface for connecting with the Spy component. Lines 13-17 represent the Client component declared in the original FTP model, lines 14-16 show the new interface added to the Client component. Lines 19-20 represent the binding to connect the Client with the Spy component.

AOP allows us to introduce new code into objects without the objects needing to have any prior mechanism of that introduction. The FTP model has been validated and we assume we don't have the source code so we cannot change its implementation to introduce some concerns about spy-ware. The listing 4.4 shows how using AOP we can add new concerns (Here, a spyware that sends information via a dedicated component interface) inside an existing model.

Line 01 explains we want to intercept a method call, and do something before the method was called. Line 02 shows methods we want to intercept: all methods from the FTPService java interface called by a ClientImpl class. Line 03 adds a condition, only component bound with a Spy component are concerned. Line 05 asks the Client interface connected to the Spy component to have this one. Line 06 calls 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.

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

Listing 4.4: AspectJ code used to inject spyware functionnality related to layout of figure 4.5.

```
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 \}
```

4.2.4 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].

4.3 SoC and Reuse in Simulation Engine

The Simulation Engine is the core part of the computer simulation program. It provides the run-time services and algorithms used to execute the Model (compute its Trajectory) and observe its outputs. In [Barr, 2004], Barr distinguishes three main techniques for providing these services and algorithms: by means of Kernels that operate similarly to an Operating System kernel and provide a transparent mechanism to run simulation based on unmodified programs, or by means of Libraries, that offer less transparency in exchange of performances, or by means of dedicated programming languages that simplifies the development of simulations but suffer from specialization effects.

The simulation engine executes simulation-runs. Each simulation-run corresponds to the computation of a Trajectory.

As previously stated, the processing of an event normally leads to changes in the system

state. It may also lead to the emission of new events. Therefore, causal relationships exist between events, and the simulation engine should offer a minimal set of mechanisms to preserve these causality chains. The engine must ensure a partial order of the processing of events according to their time of occurrence:

- The simulated time changes when all the events occurring at the same time have been processed.
- New events generated in response to a given event may not be associated with an occurrence time prior to the current simulated time (the occurrence time of the currently processed event)
- The new simulated time may only advance to the date associated with the closest event in the future (the event with the closest associated occurrence time from the current simulated time)

Separating the engine from the model is a common practice, which allows to reuse the same engine implementation with various models. However, once a model is developed for a particular engine, it is often difficult to move the same model to another engine, because the model often depends on and makes heavy use of unique features provided by the engine. A well known case is that of the NS-2 simulator that involves both a scripting language (OTcl) and C++ and forces the model developer to add instructions in this C++ code to deal with the scripting level.

DEVS has been the first to introduce the concept of abstract simulator. This concept allows to decouple the model from the simulator and to choose an engine implementation independently from the model. We propose a technical solution for clearly separating the engine from the model. Our solution enables to follow the principles mentioned in DEVS. Separation of concerns allows to test new simulation algorithms easily with a simple replacement of the engine. We can choose the most appropriate engine for a given model to have the best performance. We could also implement new formalisms into the same simulation environment and turn OSA into a testbeds for simulation formalisms or algorithms.

The next section describes the design of a new engine and its integration into our prototyping platform: OSA.

4.3.1 Case Study: OSA Simulation-Engine

As mentioned in section 2.3.2, our experimental platform OSA is based on top of the FRACTAL component model. In FRACTAL, the non-functional part of the application must take place in the membrane (the content part of the component host the functional part of the application). Now, if the model is the functional part of the simulation application, the code needed to run the simulation model becomes the non-functional part and must take place in the membrane as simulation controllers. Benefits of doing this is that all components that use these membranes have the same simulation controllers. Thus, the simulation engine is distributed over all components that are equipped with a

control membrane implementing the simulation non-functional services (figure 4.6). This membrane will be referred to, in the remainder of this section, as the **primitiveSim** membrane. A second membrane (**compositeSim**) is also available for composite simulation components.

The services of the primitiveSim and compositeSim membranes are accessed through a dedicated `simulation-controller` interface. Component with or without `simulation-controller` interface can possibly be mixed: components with a `simulation-controller` interface are called *active components* and those without such an interface are called *passive components*. Indeed the simulation services provided by the `simulation-controller` interface allow the component to wait or resume on some event or condition. Therefore components with this interface must implement a scheduler and manage threads activity and concurrency.

Hereafter, we first describe one simulation service offered to the component and then we discuss how to implement (or replace) the OSA simulation engine. This API provides a process-oriented simulation service.

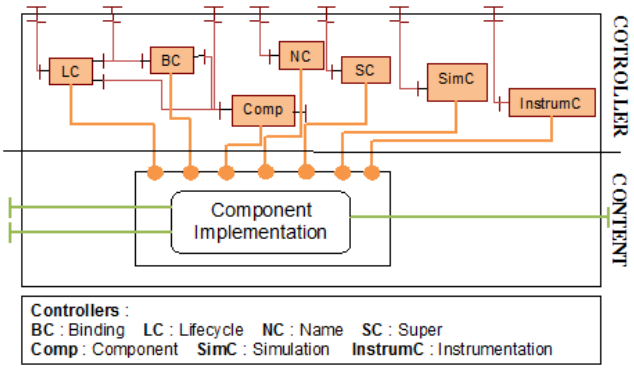

Figure 4.6: Anatomy of an OSA component.

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

- `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[C.A.R. Hoare, 1978]; optionally, the waiting may be guarded with a time-out 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[Bovet et al., 2002]

- `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.

This process-oriented design allows to implement models that follow the simpler event-oriented paradigm. For example, a "call-back-on-event" mechanism such as the one found in ns-2 can easily be emulated using a single component with a single thread, and by calling the schedule-myself() method to schedule the callback of that component's methods (the only constraint is that the callback methods must be listed in the component business interface). The synchronisation mechanism provided by the wait()/release() methods allows multiple threads to execute concurrently with safe interactions. The spin_lock() method provides the necessary support for safe reentrance in case of parallel multithreaded execution. As an example, we provide in listing 4.5 the code that could be used to execute a Basic DEVS model. We assume that delta_ext(), delta_int(), lambda(), ta() and init_state() are virtual methods provided by the DEVS model (init_state() computes the initial state of the model at time t=0)

Our DEVS emulation component provides an input() method to receive inputs from other components; it uses an output() method to send outputs to other components.

However, thanks to the flexible design of OSA and FRACTAL, 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 share the same API which means that, theoretically, components developed for various simulators may inter-operate in the same simulation scenario.

Implementing a simulation engine in OSA As already mentioned in previous section, the simulation engine implementation is located (distributed) in the `simulation-controller` implementations that lay in each active component. Unfortunately FRACTAL does not support the notion of shared component between membranes. Although it is not forbidden by the specification, none of the FRACTAL implementation we have tested offer this possibility. To share a scheduler (located in the membrane) between different components, simulation controller in the membrane is connected to an external component (the scheduler), which is shared between components of the model.

Implementing a new simulation engine for OSA mainly consists in developing and replacing all or parts of the `simulation-controller` implementation. As shown on figure 4.7, the current `simulation-controller` implementation is built using three types

Listing 4.5: Code to execute a Basic DEVS model.

```
private  int input_wq;  // An arbitrary object used for synchonization

void input(message_bag msg) {
    release(input_wq, msg);
}

void OSA_DEVS_run(){
    state = init_state();
    sigma = ta(state);
    last = current_time();
    while(true) {
        message_bag input_msg = wait(input_wq, sigma);
        if (input_msg == TIMEOUT_MSG) {
            output_msg = lambda(state);
            state = delta_int(state);
            sigma = ta(state);
            output(output_msg);
            last = current_time();
        } else {
            elapsed = last - current_time();
            last = current_time();
            state = delta_ext(state,elapsed,input_msg);
            sigma = ta(state)
        }
    }
}
```

of abstractions, each having their own interface: the event queues, the event scheduler, and the explicit simulation services. Because these abstractions have their own interfaces, their implementations may be replaced (rewritten) independently. In addition, these `simulation-controller`s, that are distributed in all the components, use two centralized objects: the `SimEngine` for boot-strapping the simulation and the `super-scheduler` which ensures the synchronization of the local schedulers of the active components.

4.3.2 Simulation Concerns in ADL

The simulation engine is the very core part of the architecture. The simulation engine executes scenarios. During a scenario execution, also called a simulation-run, the simulation engine computes the history of a system model given its initial state and a set of exogeneous events. An event is a stimulus occurring at (associated to) a particular date in the simulated time. We simply call this particular date the occurrence time (of the event). An exogenous event is an external stimulus applied to the system model at a given occurrence time (one exogeneous event at least is required to start the simulation). In response to these exogenous events, the system model may (and usually does!) produce endogeneous events. An endogeneous events corresponds to an internal stimulus occurring at a given occurrence time.

Following the principle of separation of concerns, we want to add in a separate layer the data about exogenous events. Listing 4.6 is an example of architecture layer containing elements that can be added to the ADL. Line 4 defines the architecture name. Line 6 describes the component name in which we want to add exogenous events. Lines 6 to 11

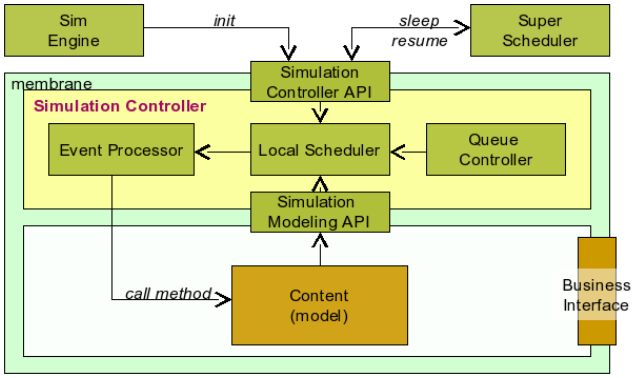

Figure 4.7: Internal architecture of the simulation-controller.

correspond to exogenous events we want to declare:

- exoevents : this tag opens a block in which several exogenous events may be declared. It has a single signature mandatory attribute, which is an (ADL) reference to an interface of the component. In other words, the string used in this attribute must match the one given for the name attribute of an interface tag within the same component (the itfname string in the above listing).
- exoevent : this tag declares an exoevent. An exoevent corresponds to the invocation of a method at a particular time in simulation. This tag accepts the following attributes:
  - name : the event name (any string allowed)
  - type : the event type. The only supported event type for now is StartOfCall.
  - time : the time in simulation at which the event is to be scheduled. The string value MUST be translatable to a positive floating point number.
  - method : in OSA, all event are associated to method calls. This string is the name of the (Java) method to be invoked, without specifying the interface: this method MUST belong to the interface declared in the surrounding exoevents tag.
  - param : a string parameter to be provided to the previous method. There is currently a limitation on the signature of the methods that can be used to declare exogeneous events: the method MUST return void and MUST accept a (unique) String parameter. More general forms of method signatures will probably be supported in future versions.

This definition is meant to overload the layer containing the definition of the Hello component.

We presented in section 2.4.2 FRACTAL ADL as an extensible and modular toolchain framework for reading architecture descriptions. This toolchain can be extended to suit our

Listing 4.6: Scenario defining exogeneous events.

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition ... >
03
04<definition name="fr.inria.osa...scenario1">
05
06  <component name="Hello">
07    <exoevents signature="itfname">
08      <exoevent name="Foo" type="StartOfCall" time="10.00" method="SendHello"
                  param=""/>
09      <exoevent name="Bar" type="StartOfCall" time="25.50" method="SendHello"
                  param=""/>
10    </exoevents>
11  </component>
12
13</definition>
```

needs. To accomplish that, new XML tags or attributes can be added to the FRACTAL ADL DTD grammar description, and the ADL processing toolchain can be extended to treat them.

Integrating "exoevents" in FRACTAL ADL required to modify the grammar but also the Loader component of FRACTAL ADL [Quema, 2005] to integrate the new Exoevent Loader in the toolchain of the FRACTAL ADL factory. The Loader component transforms FRACTAL ADL definition into an abstract syntax tree (AST). Every component of the toolchain takes the AST loaded by the delegated component,makes modifications on the AST, and then returns AST to the next component in the toolchain. Therefore, our Exoevent Loader component is placed in last position after components have been loaded by the Loader component. The Exoevent Loader adds exogenous event to the queue of the simulation controler located in the membrane of the simulation component.

We have described how to add a new tag in FRACTAL ADL that will be processed and delivered to a controller located in the membrane of a component. This solution can be used to replace or add other tags in FRACTAL ADL, such as tag for instrumentation to define which variable to probe, or tags for a dedicated formalism (change the "interface" tag of FRACTAL ADL to add the notion of bandwidth for example).

4.4 SoC and Distribution of Large Scale Simulation

Large scale simulations sometimes need several computational nodes. Reuse can lead to the composition of simple models not designed for distributed execution. In this case the distribution of the simulation requires to modify the architecture of the model to establish remote connections. These modifications can not occur within the existing architecture or we loose the benefits associated with reuse. Separation of concerns requires to declare the existence of remote connections in a separate layer that will be added to existing layers that define the architecture of the model. Moreover, we can not change the model code to support these external communications or we loose the benefits associated with reuse. Encapsulation of the model code in components allows remote connections to be

Listing 4.7: Deployment architecture using FRACTAL RMI

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition ... >
03
04<definition name="fr.inria.osa.DistributeNodeServer">
05
06      <component name="Node">
07              <virtual-node name="remote-node1" />
08      </component>
09
10      <component name="Server">
11              <virtual-node name="remote-node2" />
12      </component>
13
14</definition>
```

managed by the FRACTAL component model. The code of the model remains the same regardless of the distribution. The distribution must be transparent to existing models and architecture. In this section we propose two solutions to distribute a simulation based on FRACTAL tools: FRACTAL RMI and FRACTAL BF.

4.4.1 FRACTAL RMI

FRACTAL RMI allows to distribute a FRACTAL architecture through the RMI protocol. FRACTAL RMI handles the distribution of components and communications seamlessly. Every interface of a FRACTAL component is a remotely accessible access point. The FRACTAL ADL language provides the <virtual-node> tag to specify on wich computational node components are distributed.

Listing 4.7 shows a layer of architecture that allows to distribute the architecture defined in listing 4.11 of the previous section. Line 4 describes the name of the definition. Lines 6-8 describe the Node component. Line 7 associate the Node component to the virtual node “remote-node1”. This layer deals only with remote connections and it is normal (and recommended) that we find no further information on components due to the separation of concerns. Lines 10-12 describe the Server component. Line 7 associate the Server component to the virtual node “remote-node2”. During the execution of the simulation, we associate the variables "remote-node1" and "remote-node2" to computational nodes (for example remote-node1 will be associated to 192.168.0.1 and remote-node2 to 192.168.0.2). This layer of architecture associated with the architecture defined in listing 4.11 forms a distributed architecture where components Node and Server are running on remote machines. By simply adding this distribution-layer, it is not necessary to modify the existing architecture or model code. The disadvantage of FRACTAL RMI is that the interpretation of the architecture is on a single machine and the representation of the architecture should fit in memory in order to start the instantiation. It is a limiting factor because on very large architecture it is impossible to interpret the overall architecture on a single physical machine. In this case and to see components as remote services, we propose the use of FRACTAL BF.

Listing 4.8: A distribution layer that provides a rmi service

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition ... >
03
04<definition name="fr.inria.osa.DistributedServer"
05              extends="fr.inria.osa.Server"
06              arguments="address,port">
07
08      <exporter type="rmi" interface="Server.link">
09              <parameter name="serviceName" value="link" />
10              <parameter name="hostAddress" value="${address}" />
11              <parameter name="port" value="${port}" />
12      </exporter>
13
14</definition>
```

4.4.2 FRACTAL BF

The FRACTAL Binding Factory (FRACTAL BF) transforms a component into a remote service, or connects a component to a remote service. FRACTAL BF allows the use of various protocols such as web services, RMI, and much more. It is also possible to implement new communication protocols into FRACTAL BF. To do this, FRACTAL BF provides a simple API or new tag in FRACTAL ADL. We call “exporter” the entity that will transform a server interface into a remote service. we call “binder” the entity that will connect a client interface to a remote service.

The reference model is the model described in listing 4.11. We want to split this model in two parts. Part “Server” which offers a service, and part “Node” which requires a connection to this service. Layer architecture described in the listing below 4.8 transforms the component “Server" in a service accessible via the RMI protocol. Then listing 4.9 connect the component Node to the remote Server service.

Listing 4.8 describes the architecture of the model “Server” that we want to deploy and make it accessible remotely. Line 4 describes the architecture name, line 5 describes the existing architecture it extend and line 6 describes the parameters used. Lines 8-12 correspond to FRACTAL BF and transform the interface “link” of component “Server” in an interface accessible remotely. Line 8 describes the protocol type (in this case, RMI) and the interface that we want to make remotely accessible (here, the interface “link” from component “Server”). Line 9 describes the RMI service name we want to associate with this binding. Line 10 describes the address of the local component Server. Line 11 describes the communication port.

Listing 4.9 describes the architecture of the model “Node” that we want to deploy and to connect to the remote “Server” service. Line 4 describes the architecture name, line 5 describes the existing architecture it extend and line 6 describes the parameters used. Lines 8-12 correspond to FRACTAL BF and transform the interface “link” of component “Node” in a remote connection to a remote service. Line 8 describes the protocol type (in this case, RMI) and the interface that we want to connect to remote service (here, the interface “link” from component “Node”). Line 9 describes the RMI service name we want to use for this binding. Line 10 describes the address of the remote component Server.

Listing 4.9: A distribution layer that requires a rmi service

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition ... >
03
04<definition name="fr.inria.osa.DistributedNode">
05             extends="fr.inria.osa.Node"
06             arguments="adress,port">
07
08      <binder type="rmi" interface="Node.link">
09              <parameter name="serviceName" value="link" />
10              <parameter name="hostAddress" value="${address}" />
11              <parameter name="port" value="${port}" />
12      </binder>
13
14</definition>
```

Line 11 describes the communication port.

With FRACTAL BF, we cannot reuse as is a complete architecture and deploy it. But we can recreate the architecture using several smaller architecture that will be connected together at runtime. The benefits of this approach are that this architecture are smaller and require fewer resources to be instantiated and architecture can be instantiated in parallel. Moreover, FRACTAL BF allows to establish remote connections without modifying the code of the model. However, it is difficult to have a global view of the simulation as we instantiate and run several architecture in parallel on several computational node.

4.5 Other Means for Enforcing Reuse

In this section, we present first time how the use of dynamic architecture allows you to create architectures that are more easily reusable. In a second step, we present the Maven project management tool which promotes the replayability and the reuse of projects.

4.5.1 Promote Reuse With Dynamic Architecture

The shared component is a means for reuse: a shared instance appears in several locations in the architecture definition [Dalle et al., 2008, Dalle, 2006]. Another way to reuse is to have only one defined component in the architecture definition but several instance. This last feature is interesting to describe dynamic architectures. FRACTAL ADL provides an interesting feature to describe dynamic applications: the couple template/factory. We can use this to favor reuse (by encouraging the creation of generic architectures) creating a loop mechanism within FRACTAL ADL. Take the example of an application containing multiple nodes connected to a server. Since FRACTAL ADL has no way to express the notion of loop, the most elegant solution is to describe the complete architecture as shown in listing 4.10. Line 4 describes the name of the architecture. Lines 6-7, 9-10, and 12-13 respectively describe the 3 components Node1, Node2 and Node3 whose definition are given in the file Node.FRACTAL. The definition described in this file represents a primitive component having a client interface named “link”. Lines 15-16 describe the Server

Listing 4.10: Model architecture without loop

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition ... >
03
04<definition name="fr.inria.osa.NodeServerExample1">
05
06      <component name="Node1"
07              definition="fr.inria.osa.Node"/>
08
09      <component name="Node2"
10              definition="fr.inria.osa.Node"/>
11
12      <component name="Node3"
13              definition="fr.inria.osa.Node"/>
14
15      <component name="Server"
16              definition="fr.inria.osa.Server"/>
17
18      <binding client="Node1.link" server="Server.link" />
19      <binding client="Node2.link" server="Server.link" />
20      <binding client="Node3.link" server="Server.link" />
21
22</definition>
```

component whose definition located in the file Server.FRACTAL represents a primitive component with a server interface named "link". Finally lines 18-20 describe the connection between the 3 components Node1, Node2 and node3 with component Server.

The reuse of this architecture is complicated if we want to vary the number of Nodes in the application. FRACTAL allows through factory and template to describe only one template component Node and bind it to a Factory component that will take care of duplicating it. Unfortunately, the template feature does not permit to a component to behave as a regular one. Thus, it is imperative to add the logic in the Factory component so it can bind new cloned components (in our case, bind a cloned Node to the Server). We believe this is an obstacle to the separation of concerns that can have many adverse effects on reuse. Indeed, a dynamic architecture can be reused by another architecture, and FRACTAL ADL features such as inheritance and overloading allow to modify the binding between components as described in section 4.2. In this case, we must also replace the Factory component by a new Factory component to ensure the correct binding between cloned component and the rest of the architecture. A template is not a component that behave as a regular one, it's a component only used as a model for cloning. Thus, it is not possible to establish bindings between a template component and regular component. Therefore, it is impossible to reuse an architecture in order to add dynamicity, and it is mandatory to add knowledges about bindings in the Factory component. To cope with these limitations in the existing FRACTAL ADL implementation, we propose a solution to add dynamicity to existing architectures while suppressing the necessary the logic in the Factory component to establish bindings.

Our solution consists in changing the behavior of a template so that it can also behave as a component. Thus, the factory responsible for duplication of this template can create new instances, but also bind new instances in the same manner as the original template

Listing 4.11: A simple model layer

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition ... >
03
04<definition name="fr.inria.osa.NodeServerExample2">
05
06      <component name="Node"
07              definition="fr.inria.osa.Node"/>
08
09      <component name="Server"
10              definition="fr.inria.osa.Server"/>
11
12      <binding client="Node.link" server="Server.link" />
13</definition>
```

and therefore does not need additional knowledge.

Listing 4.11 shows an example of a description that we want to reuse to add dynamicity. Line 4 describes the architecture name. Lines 6-7 describe the Node component whose description is defined in the file Node.FRACTAL. The definition described in this file represents a primitive component having a client interface named "link". Lines 9-10 describe the Server component whose definition located in the file Server.FRACTAL represents a primitive component with a server interface named "link". Line 12 describes the connection between the Node and Server components through their interface "link".

We would like to reuse this architecture, but we want to be able to vary the number of Node components. Figure 4.8 represents a schematic view of our solution. Each different component are represented by a geometric sign. The circle represents the Node component (the N represents the content of the Node component). The triangle represents the Server component (the S represents the content of the Server component). The star represents the Duplicator component (the D represents the content of the Duplicator component). In a layer, the Node component is connected to the Server component. In another independent layer, the Duplicator component is connected to the Node component. In this layer, the Node component don't have any content but is declared as a template (the T mentioned on the top of the circle). The composition of this 2 layers is the result shown in the bottom part of the figure. The Duplicator component is connected to the Node component which is a regular component with template feature. The Duplicator component duplicates the Node component and bind the new Node component as the template one.

We propose to describe an additional layer that will be based on this architecture. Listing 4.12 describes the reuse of the previous definition to which we add the possibility to have a variable number of Node components. Lines 4-6 describe the architecture name, which architecture it extends, and defines the "scale" parameter which is the number of Node components we want in our architecture. Lines 8-10 describe the Node component which is added the template possibility. Lines 12-14 describe the factory component "Duplicator" whose definition is described in the file Duplicator.FRACTAL. This definition describes a primitive component with a client interface "template". We see in line 13 that the parameter "scale" is passed to the factory component Duplicator. Line 16 established a link between the interface "template" of the factory component Duplicator and the

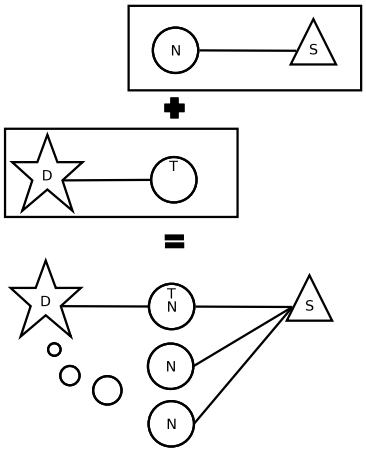

Figure 4.8: Schematic view of a dynamic architecture.

Listing 4.12: Model architecture with loop

```
01<?xml version="1.0" encoding="ISO-8859-1" ?>
02<!DOCTYPE definition ... >
03
04<definition name="fr.inria.osa.NodeServerExample3"
05            extends="fr.inria.osa.NodeServerExample2"
06            arguments="scale">
07
08      <component name="Node">
09              <template-controller desc="simPrimitiveTemplate" />
10      </component>
11
12      <component name="Duplicator"
13              definition="fr.inria.osa.Duplicator(${scale})">
14      </component>
15
16      <binding client="Deplicator.template" server="Node.component" />
17
18</definition>
```

interface “component” that exists in all FRACTAL components. At runtime, during the initialization, the factory component Duplicator will try to duplicate “scale” times the component connected to its “template” interface. Thanks to our changes on FRACTAL, we can declare a component as a template too. The factory component Duplicator will duplicate and then bind new components in the same way that the original template component, i.e. bind all new Node components to the Server component. This generic solution duplicates and binds in the same way component bound to the “template” interface of the factory component Duplicator.

Thus, we provide a solution to the problem of loop in FRACTAL ADL, which requires only minor changes to FRACTAL. This change has to my knowledge no adverse side effect and may be proposed to the FRACTAL developer community. A study of the effects on performance during the instantiation of large architecture with and without loops is seen

in chapter 7.

4.5.2 Enforcing Reuse and Replayability with Maven

Once the software is ready for reuse, a last element is required to help dealing the issues related to the dissemination of the reusable code. For this purpose, we use the Apache Maven project, which is a project management and comprehension tool, developed by The Apache Software Foundation. Apache Maven is a free software tool for managing and automating production of Java software. The goal is comparable to Make under Unix: produces a software from source, optimizes the tasks necessary for this purpose and ensures the production order.

Maven uses a paradigm known as the Project Object Model (POM) to describe a software project. Each project or subproject is configured by a POM that contains the information necessary to process the Maven project (project name, version number, dependencies to other projects, libraries needed to compile, names of contributors, and so on). This POM is materialized by a pom.xml file at the root of the project. This approach allows the inheritance of properties of the parent project. If a property is overridden in the POM project, it covers those who are defined in the parent project. This allows to reuse configuration.

Another important feature and relatively specific to Maven is its dependency management. Indeed, Maven can automatically download the dependencies of a project. Maven can also publish a project on a repository in order to make it available to other projects. The Maven dependency management is simplified by the notions of inheritance and transitivity. Maven allows the use of different categories of repositories:

- **local repository** includes everything that the developer has used and developed.
- **global repository** includes everything that have been publish publicly by the developers.
- **enterprise repository** makes available to all developers of a company private projects.

The use of these repositories allows versioning, and thus promotes replayability by tracing and downloading all the dependencies for a given version of a project.

Maven provides a plugin architecture for adding new features. These plugins are available on Maven repositories, or can be developed. Thus, Maven can be configured to suit our need.

Comparing to existing tools to support replayability and reuse, maven is a all-in-one tool. It allow to start a new simulation project, manage dependency, compilation and execution processes, and most of all, put everything in a repository for a further reuse. Moreover, Maven is not dedicated to a specific software.

Listing 4.13 represents the a POM file of a simulation experiment. Lines 3-8 indicate the name, type and version of the project. Lines 10-16 indicate the enterprise repository we want to use to access private dependencies. Lines 20-41 show the dependencies needed

Listing 4.13: A simple POM file of an OSA experience

```
<?xml version="1.0" encoding="UTF-8"?>
01<project ... >
02
03      <modelVersion>4.0.0</modelVersion>
04      <groupId>fr.inria.osa.helloworld</groupId>
05      <artifactId>experience</artifactId>
06      <packaging>jar</packaging>
07      <name>OSA helloworld experience</name>
08      <version>1.2.3</version>
09
10      <repositories>
11              <repository>
12                      <id>osa.inria.fr</id>
13                      <name>OSA Repository</name>
14                      <url>http://osa.inria.fr/osa_repo/</url>
15              </repository>
16      </repositories>
17
18      [...] <!-- plugins configuration -->
19
20      <dependencies>
21              <dependency>
22                      <groupId>fr.inria.osa.helloworld</groupId>
23                      <artifactId>model</artifactId>
24                      <version>1.2.3</version>
25              </dependency>
26              <dependency>
27                      <groupId>fr.inria.osa.helloworld</groupId>
28                      <artifactId>instrumentation</artifactId>
29                      <version>1.2.3</version>
30              </dependency>
31              <dependency>
32                      <groupId>fr.inria.osa.simulationEngines</groupId>
33                      <artifactId>basicEngine</artifactId>
34                      <version>1.7</version>
35              </dependency>
36              <dependency>
37                      <groupId>fr.inria.osa.helloworld</groupId>
38                      <artifactId>scenario</artifactId>
39                      <version>1.2.3</version>
40              </dependency>
41      </dependencies>
42
43</project>
```

to run this project. This experience is the composition of many simulation projects: a model (version 1.2.3), a scenario (version 1.2.3), an instrumentation (version 1.2.3), and a simulation engine (basicEngine version 1.7). The model project defines dependency to a simulation API (basicAPI version 1.0) that the simulation engine (basicEngine version 1.7) must implements.

4.6 Conclusion

In this chapter, we have shown how separation of concerns allows to promote reuse. We have shown practical examples of the use of techniques and software engineering tools such as component programming and aspect-oriented programming. We have shown how the use of dynamic architectures allow to create architectures more easily reusable. We have also shown that the use of the Maven project management tool helps with the dissemination of the code and favors replayability.

The set of techniques presented in this chapter applies as seen throughout this chapter on the layers of modeling, scenario and deployment, but these practices should apply to all concerns of the simulation such as instrumentation, simulation, and potentially other layer following concerns relating to study that one wishes to lead.

Chapter 5

Contributions to Instrumentation

Contents

This chapter was written in collaboration with Denis Conan, Sebastien Leriche and Olivier Dalle and has been published in the third International ICST Conference on Simulation Tools and Techniques in Malaga, Spain [Ribault et al., 2010].

The workflow used for studying a system using discrete-event simulation is often described in the simulation literature, e.g. in [Banks et al., 2004, Law, 2007]. 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 [Zeigler, 1984], 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 to 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 [Dalle and Mrabet, 2007], Dalle and Mrabet already presented the OSA Instrumentation Framework (OIF). 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 chapter 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 to be able 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 a 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 processors 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 at the bottom-half of the framework to build the probes in charge of collecting data samples. At the top-half of the framework, we (re)use the COSMOS framework [Conan et al., 2007, Rouvoy et al., 2008] which allows for on-line computations (e.g. statistics) and data transmission and storage (notice COSMOS is not a contribution of this thesis). 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 [Varga and Hornig, 2008] to post-process simulation data.

In the sequel of the chapter, section 5.1 presents our motivations to build a generic open instrumentation framework and how we plan to achieve our goals. Then, section 5.2 presents the Open Simulation Instrumentation Framework. Finally, section 5.3 concludes the chapter and draws some perspectives.

5.1 Motivations and Objectives

Law presents in [Law, 2007] a general workflow to build a valid and credible simulation study. Figure 5.1 summarises Law's workflow, focusing only on tasks where instrumentation is needed. The first four tasks of 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, ran and analyzed according to the simulation objectives. The conclusions drawn from the simulation results are finally presented in a document.

In this chapter 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. 5.1.1), the ability to apply the same validation process to real and simulated systems (sec. 5.1.2), the ability to easily switch from live to post-mortem analysis (sec. 5.1.3) and the need for composition in data processors (sec. 5.1.4).

5.1.1 Separation of Concerns

To design the instrumentation of a simulation model by following the workflow depicted in Fig. 5.1, most simulators offer a simulation API for the declaration of observable data within the simulation model. This common practice implies 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

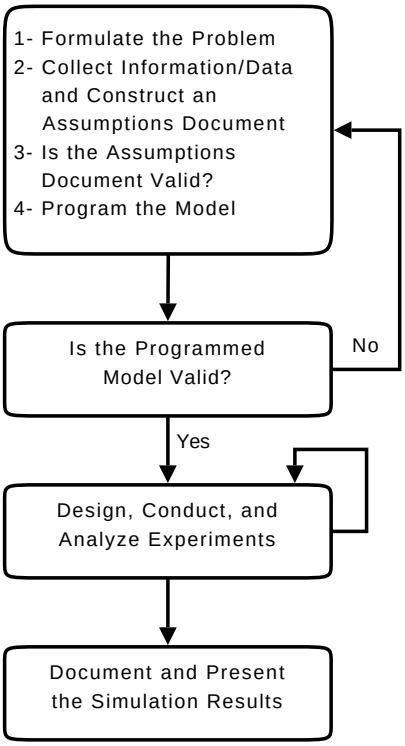

Figure 5.1: Simulation workflow focusing on instrumentations tasks.

provides many benefits. For example, keeping the simulation model clear from any instrumentation allows to reuse it in many different 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 give enough flexibility to instrumentation experts so they can work independently from each others or from modeling experts. This allows you to program the model (item 4 of the top box of figure 5.1) independently from the validation tool.

We propose to use the Aspect-Oriented Programming (AOP) paradigm [Kiczales et al., 1997] 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 [Conan et al., 2007, Rouvoy et al., 2008].

Figure 5.2 shows schematically the separation of concerns between the different layers of the instrumentation. COSMOS is based on FRACTAL, we can use the same properties as the one we used for the separation of concerns between models or scenarios. Thus, the instrumentation may be the composition of several layers of instrumentation through the

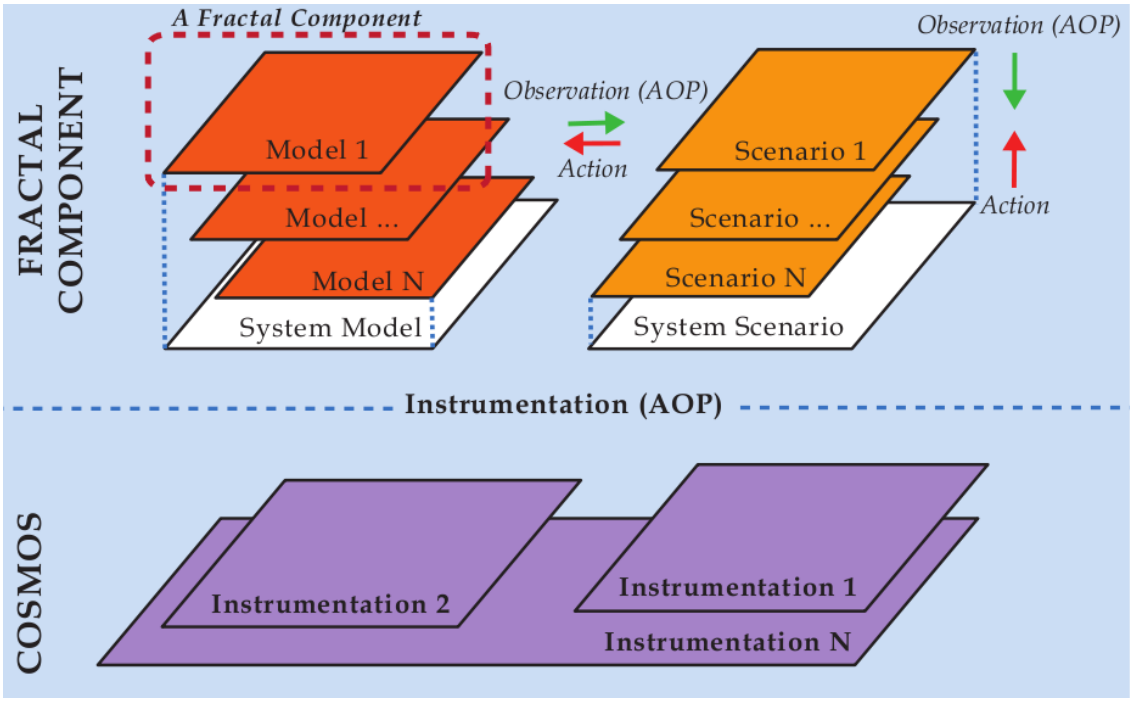

Figure 5.2: A zoom on the instrumentation part with COSMOS of the OSA layered approach.

mechanism of inheritance and overloading of FRACTAL ADL. We can define one or more layers of instrumentation that monitor the scenario (Instrumentation1), the model (Instrumentation2) or overload some of the existing instrumentation to continue the computation made on the data (Instrumentation N).

5.1.2 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 5.1. The validation process can help to validate a simulation model by comparing simulation results with data collected during 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. In addition, COSMOS can be used for simulation purposes, which allows to reuse the same data processors when experimenting the real system and

simulating the virtual system.

5.1.3 From Live to Post-Mortem Analysis

The third box of figure 5.1 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 5.1).

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 better quality results. As a consequence, the data flow can be optimized. Moreover, it is easier to replay a study that integrates its data processors. Indeed, since statistical analysis are done 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.

5.1.4 Data Processors Composition

Another interesting feature is the possibility to write simple instrumentation and data processors (validation of simulation results and analysis) and combine them into more complex ones. This approach helps reducing the complexity of designing an instrumentation since writing many simple instrumentation and data processors is easier than writing a complex one. Furthermore, considering reuse, it is more likely to reuse several times simple and generic data processors than to reuse a complex one. Another case of reuse is the design of a new data processors from an already existing one from a catalogue. Reusing and composing data processors is also an asset when comparing studies sharing the same data processors. In that case, it is easier to compare and trust the results because vali-

dation of simulation results and results analysis are the same among studies. Of course, since instrumentation makes the bridge between simulation model and data processors, 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 processor is independent and can be reused regardless of the language and the simulation model. Thus, in worst case it will be necessary to create/adapt the instrumentation code, but we can always reuse the data processors.

5.2 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 5.1 with OSIF.

5.2.1 COSMOS

COSMOS (COntext entitieS coMpositiOn and Sharing) [Conan et al., 2007, Rouvoy et al., 2008, Romero et al., 2009] is a LGPL component-based framework for managing context data in ubiquitous applications; it is based on the FRACTAL component model [Bruneton et al., 2006]. 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 processors, 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 processors 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 [Leclercq et al., 2005].

5.2.2 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 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 5.3 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 5.3 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.

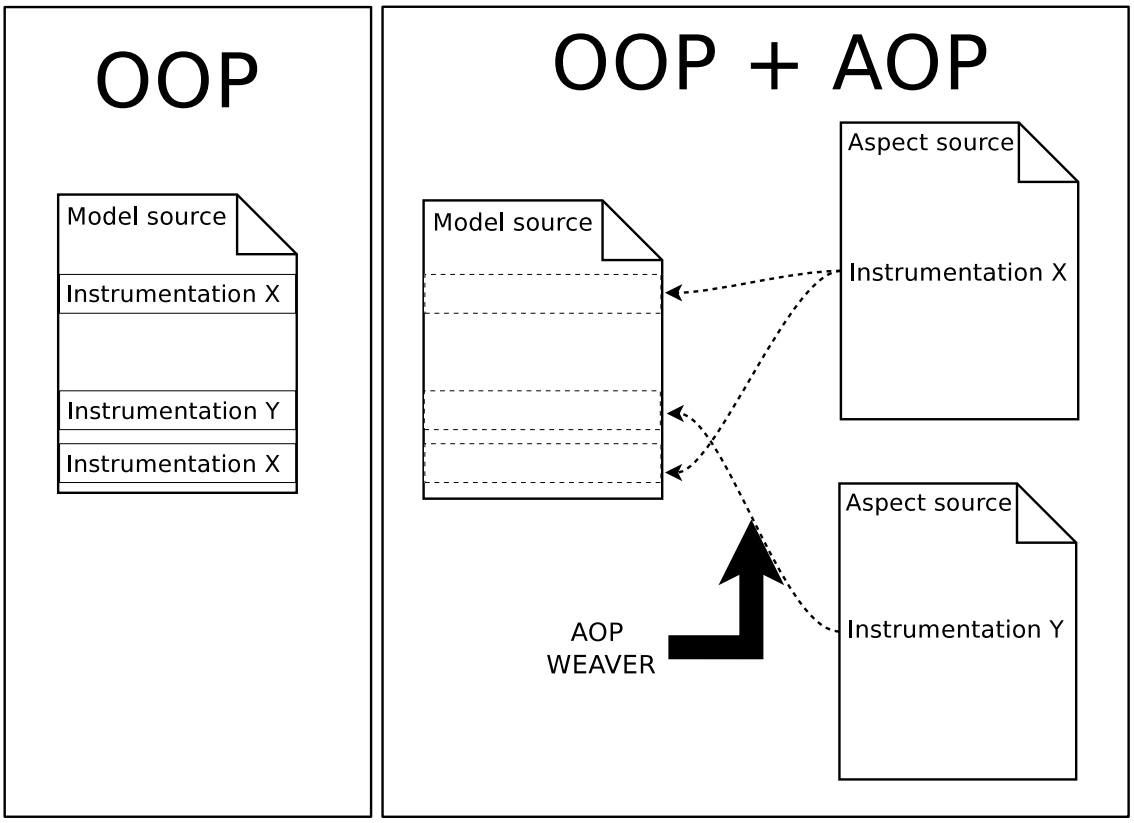

Figure 5.3: Separation of concerns using AOP.

As an example, listing 5.1 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 5.3. 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 5.1: 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");
  }
}
```

Listing 5.2: 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");
  }
}
```

Thanks to AOP, it is now possible to separate the modeling concern from the instrumentation concern. The aspect of listing 5.2, written in AspectJ, shows how to isolate the instrumentation concern in a separate module as illustrated on the right part of figure 5.3. The aspect peer_instrumentation calls the instrumentation framework right after the execution of methods boot() and halt().

Therefore, listing 5.3 illustrates the same modeling code stripped from instrumentation concern.

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 processors can be developed independently by instrumentation experts and reused more easily.

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. As part of COSMOS, the data collectors are connected to sensors such as ,for example, heat sensors, or sensors that

Listing 5.3: Java class with separation of concerns.

```
public class Peer{
  String name;

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

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

analyze the network load. As part of the M&S, environment is the model to simulate. The probes are then implemented through AOP. 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.

5.2.3 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 during execution 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 output either by data collectors or other data processors. Therefore, data processors are organized into hierarchies with the possibility of sharing. A data processor (a node of this graph) can be parameterized to be passive or active, blocking or not blocking in observation or in notification, as explained hereafter:

- **Passive or active** A passive node obtains simulation data on demand; a passive node reacts to explicit requests made by other nodes. An active node is associated to a thread and initiates the gathering and/or the treatment of simulation data on its own. The thread may be dedicated to the node or be assigned from a pool. Typical examples of an active node are the centralization of several types of simulation data, the periodic computation of a higher-level simulation data, and the transmission of the latter information to upper nodes. An active node can be used to explicitly control the sampling rate of data and limit the computation overhead.

- **Observation or 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 blocking** During an observation or a notification, a node that processes the request can be blocking or not. A blocking node in observation replies to the request of the parent node or the client application by providing the most up-to-date simulation data that it possesses without requesting child nodes while a not blocking node in observation 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. A blocking node in notification computes a new observation report with the new simulation data just being notified without notifying parent nodes while a not blocking node in notification computes a new observation report with the new simulation data just being notified, and then notifies the parent node or the client application.

Figure 5.4 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.

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 can be huge. Using OSIF, we can easily build a live analysis instrumentation that directly writes only the min, max, and average lifetimes

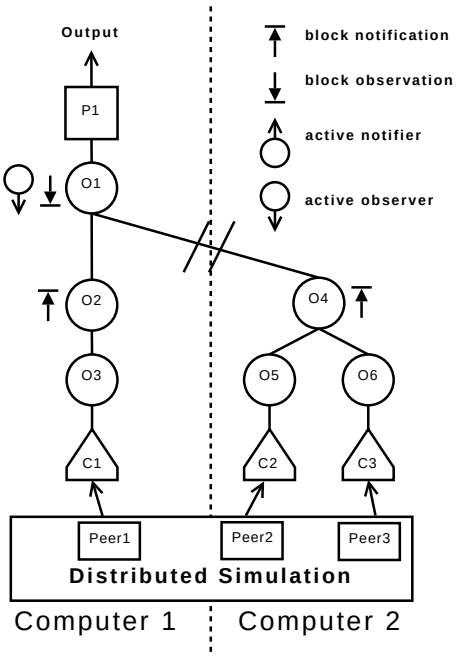

Figure 5.4: Graphical representations of data processors in a distributed simulation.

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. In other words, instrumentation policies are the "entry points" in 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 5.4. Assume we want to change our initial goal in order to keep logs of the simulation events and build a trace of the peers connections and disconnections. The node O1 aggregates and merges simulation data from nodes O2 and O4 and pushes them to the instrumentation policy P1. P1 then outputs data according to a specific format, for instance the format of the OMNeT++ analysis tool

Scave [Varga and Hornig, 2008]. Scave helps the user to process and visualize simulation results saved into vector and scalar files. So, the instrumentation policy P1 translates simulation data output by node O1 into vector or scalar files understandable by Scave, during a post analysis.

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 to reduce disk usage and network bandwidth usage. 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 data compliant with existing tools in order to compare simulation studies' results.

5.2.4 Composition of Instrumentations

In this section, we show how the architecture description language of FRACTAL (FRACTAL ADL) can be used 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 presented in section 2.4.1, and instrumentation policies are specified using FRACTAL ADL presented in section 2.4.2. 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 and sharing.

Architecture description language As presented in section 2.4.2, FRACTAL ADL is a XML language to describe the architecture of a FRACTAL application. 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 5.4 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 5.4: 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 5.5 illustrates the multiple extension capability of FRACTAL ADL. At the top, we have a FRACTAL ADL definition extending the FRACTAL ADL definition of listing 5.4 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 5.5, to merge components. The merging process allows to update and extend already defined component. Therefore, it is possible to reuse and extend existing data processors and configure it. This can range from a simple update of parameters to the replacement or addition of management contexts.

There are several benefits to using the extension and the redefinition mechanisms of FRACTAL ADL. First, writing many simple instrumentations and data processors is easier for maintenance than writing a complex one from scratch. Second, there is more chance to reuse generic simple instrumentations and data processors than a complex dedicated one. Third, it is easier to compare and have confidence in results when instrumentation and data processors 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 simulation model may come with 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.

5.2.5 From Real to Virtual System

In OSIF, we use COSMOS to process simulation data. As mentioned in section 5.1, 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 applica-

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

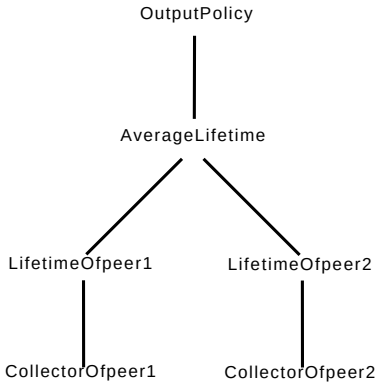

Figure 5.5: FRACTAL ADL composition mechanism and the resulting COSMOS design.

tions [Bruneton et al., 2006, Romero et al., 2009][1]. 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 is improved.

5.3 Conclusions and Perspectives

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 processors; (ii) OSIF favors validation of results by allowing to share the analysis process 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 processors in a simple way. OSIF has been successfully tested on the OSA simulator through a large case study presented in Section 7.1. OSIF has been designed so that there are no connection between the simulator, the instrumentation, and the data processors. Thus, OSIF can be used and reused on any simulator. Since AOP is available for most programming languages, OSIF could be used regardless of 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 is to federate a community around OSIF to build and share COSMOS components in order to enrich the experience of the end-users.

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

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 some unexpected side effects. A tool for describing, analyzing and verifying data processors as the 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 investigate techniques such as those described by [Himmelspach et al., 2008]. 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.

Chapter 6

Thoughts on Integration

Contents

In this chapter, we consider the problem of reusing parts of existing simulators in a new one. We started from the observation that despite no single simulation software seems to be perfect, most of the elements required to make a perfect simulator already exist as part of existing simulators. This chapter presents our solution to integrate existing simulation elements such as models and engines thanks to the ability of FRACTAL to encapsulate and compose software. It is also interesting to integrate a simulator as a back-end, particularly to reuse existing experimental planning. To demonstrate the feasibility of such integration, we use our demonstration platform OSA as front-end for the integration of existing models and engines from another simulator, but also as a back-end for integration of existing experimental planning from another simulator.

We saw in chapter 4 reasons and solutions for making tools that promote reuse. However, this reuse is limited to a community using the same tools. Integration is the ability to share and reuse software components between different user communities. Integration is thus a complex task because it must succeed in establishing means of communication between the various elements to be assembled without those being designed to work together. We also saw in section 2.3.4 that there were already solutions for integration (HLA to integrate simulation or CD++ to integrate services). We propose to extend integration to all elements of the simulation through the use of software engineering techniques such as component-oriented programming and aspect-oriented programming.

In the sequel of this chapter, section 6.1 presents our motivations to work on the integration of existing elements in simulation and how we plan to achieve our goals. Section 6.2 presents our works on the integration of parts of the JAMES II framework into OSA. Section 6.3 presents related works on the integration of elements into simulations. Finally, section 6.4 concludes the chapter and draws some perspectives.

6.1 Motivations and Objectives

By repeating the simulation activities defined in section 2.3.1, we realize that reuse can be beneficial in all stages of a simulation study. We propose to group integration techniques into 4 categories:

- **Integration of existing simulation elements** allows to integrate elements from existing simulator such as models, engine, formalism, etc. The benefits of such integration are the same as those related to reuse (see section 3.2.3).
- **Integration of existing simulation tools** allows the integration of pre and post processing tools such as experimental planning or results analysis. The integration of such tools allows to save development time, and to access reliable and powerful tools.
- **Integration of elements of the system under study** (emulation) allows to integrate within the model to simulate elements from the system that we study. There is no need to make assumptions about the behavior of the system which is integrated in the simulation. Nevertheless, we still need to make assumptions on the part of the system that we will emulate.

- **Integration of services** (mashup) allows to integrate within the simulation data or services. For example integrate within a simulated forest fire data from meteorological services to predict fire behavior. The benefits of such integration are the separation of concerns among experts and updating real-time data provided by a third partner whose job is not simulation.

Integration of existing simulation elements can be done through the encapsulation of components and separation of concerns between elements of the simulation. Integration of existing simulation tools is achieved through the establishment of communication channels between the tools, such as a common format to store data or creating a wrapper to control backend tool. Integration of elements of the system under study can be done through the encapsulation in components, separation of concerns and aspect-oriented programming that allows to intercept method call without having to modify the source code of the original software. Integration of services can be achieved through component programming. In fact, the code inside the component communicates through the interface of the component. This is the platform that implements the component model that supports the communication to remote services.

6.2 Contributions

This section presents our contributions in the integration of existing simulation elements. First, we present the integration of elements from James II into OSA. Second, we present the integration of OSA in a tool of JAMES II.

6.2.1 Integration of Existing Simulation Elements

The integration of elements of existing simulation allows to reuse existing models, scenario, engine or formalism from existing simulation software and integrate them in others simulation software. Our experimental platform based on the OSA FRACTAL component model allows to highlight our solutions based on the use of encapsulation property of component models. Thus, the following section discusses possible ways to integrate elements of the simulation framework JAMES II in OSA. As presented in section 2.3.2, JAMES II is a general and open framework based on the "Plug'n simulate" concept , which allows developers to integrate their modeling and simulation methodological ideas into, and to create their applications upon an existing framework. JAMES II currently provides over 400 plugin.

Integrating JAMES II Elements into OSA In this section, we discuss the integration of different elements from the JAMES II framework into OSA. We first present the integration of elements from the library of JAMES II such as generators or queues. Then we discuss the integration of formalism and simulation engine, such as DEVS formalism and the simulation engine associated with this formalism. Then we discuss the integration of models and different way to reuse the models (reuse of part of model or reuse of full model). Finally, we discuss the beneficial side effects that we found to integration.

Library integration JAMES II can be reused as a library with, for example, the reuse of generator or queue. There are several methods to reuse JAMES II as a library. The first is the direct use of classes of JAMES II, but this does not correspond to a strict separation of concerns. Indeed, a direct reference to classes of JAMES II prevent to easily replace them later. The second is the encapsulation of classes of JAMES II in FRACTAL components. We must then create as many components as classes of JAMES II that we want to reuse. The third is to use the plugin mechanism of JAMES II to load the implementation at runtime. We must then create a single component by type of services (random number generator, queue) that we want to reuse. During the component initialisation, the FRACTAL factory calls the JAMES II registry for an instance corresponding to the given server interface and set the singleton. However, this process can be parameterized, and thus, although using the framework functionality, we can still ask for a concrete implementation – but if this does not exist we will just return an alternative, thus here we have a softer binding between the two. We selected and implemented the third method because it minimizes the number of components required while providing as much functionality as the other two methods. Moreover, the addition of new implementations of a type of service already supported in OSA will be automatically taken into account.

Engine integration To encapsulate engines, we have done this in a similar way as the library way. In fact, to benefits from the automatic engine selection of JAMES II, we have build an engine factory component where the role is to ask the JAMES II registry for a concrete implementation according to the given model. The engine factory component is connected through the FRACTAL interface to the model component. Thus, we can reuse all the FRACTAL communication protocol to handle a communication between the simulator engine and the model.

Here, we chose not to put the simulation engine inside the component's membrane. This choice was made because JAMES II prohibits any interaction between the model and the engine (unlike the process-oriented engine of section 4.3). Therefore, there is no interest in placing the engine inside the membrane. The engine can be seen as a simulation layer on top of the simulation model whose role will be to lead state's transition (as proposed by Zeigler with his concept of abstract simulator [Zeigler, 1984]).

Model integration Reusing complete JAMES II model into OSA allow to use OSA functionalities such as the instrumentation framework on this model. To wrap a model into a component without modification, we have created a new model extending the previous one. This new model contain only the necessary information to be "componentized" - i.e. special java annotations. Fraclet[Rouvoy et al., 2006] will then add during the compilation phase some methods necessary to transform this new model (overloading the previous one) into a component. In that way, the resulting component provide a server interface corresponding to the original model interface.

Reusing part of JAMES II model into OSA allow to use OSA functionalities such as those provided by FRACTAL ADL. Moreover, it can afford to have components of references used in several simulations. To do this, without modifying the original model,

we override parts of the model that we want to transform into components. Thus, for example with the DEVS formalism, we override the classes that define the original model and we override the initialization method so that the assembly is done via FRACTAL ADL and not directly in the model code. Thus, the model is no longer a monolithic block but an assembly of components described via FRACTAL ADL. The communication protocol between components is handled by FRACTAL.

Positive effect Since it is possible to reuse elements of the JAMES II framework, it is possible to reuse development functionality of the JAMES II framework. For example, to make a new implementation of a queue, it is wise to do so within the JAMES II framework and then use it in OSA rather than implement it directly in OSA to take advantage of the queue test process of JAMES II. This separation of concern between the development framework and the runtime framework is also a form of reuse.

6.2.2 Integration of Existing Simulation Tools

The integration of existing simulation tools allows to reuse existing pre or post-processing tools and benefits from reliable and powerful tools quickly. Technical solutions to connect integrate an existing tools and requires the establishment of common rules, such as a common format for storing data or by the use of the frontend/backend pattern. We showed in section 5.2.3 page 82 that it is possible to reuse post-processing tools such as Scave (the post-processing tool of OMNeT++) through the recording of the simulation results in a format understandable by Scave.

The use of the front-end back-end pattern can be used in the case of pre-production tools such as the experimental planning of JAMES II. The uses of the experimental planning of JAMES II requires the simulation to be executed as a backend. OSA can be used from the command line and this method can be used to define the parameters of the simulation. The disadvantage of this approach is that the parameters passed on the command line are not stored in the simulation project and the replayability is not guaranteed. We used the experimental planning tool from JAMES II that saves all settings of a study. The replayability is then provided by the pre-processing tool. Another solution is the possibility to let OSA drive the experimental planning who drive the OSA simulation. The top OSA project defined with Maven using the Project Object Model ensures the replayability as it fully explicits the experiment settings.

6.3 Related Works

This section presents works on the integration of system elements and services into the simulation.

6.3.1 Integration of Elements of the System Under Study

The integration of elements of the system under study allows to reuse parts of this system and thus there is no need for any assumptions about their behavior. This saves time in

development but also in V&V.

In [Moallemi and Wainer, 2009, Holman et al., 2009], the authors show interesting technique to use the code of the model as the system code. They first develop the code for the simulation (using CD++), then use the same code into the system.

In OSA, reuse of code of the system that we study is done via the encapsulation in FRACTAL components. The integration is done without modifying the source code through the inheritance offered by Java or by using AOP (already presented in chapter 4 and chapter 5) as well as FRACTAL ADL. The integration of system element may need to intercept some calls such as system calls and emulate the behavior of the underlying system to simulate the global behavior also requires an engine with real time capabilities. Here again AOP can help to intercept and overload method call without modifying the original source code.

Notice other techniques can be used, such as changing the OS interface library using LD_PRELOAD on unix systems. In [Lacage, 2010], Lacage shows that the "use of a simple ELF dynamic loader together with replacement libraries for the Linux user space environment and the kernel space runtime environment is sufficient to allow the Direct Code Execution of unmodified user space and kernel space protocol implementations". Thus, "the same application binary can run both standalone on the host system to be used in a testbed or directly within the simulator".

However these techniques are less convenient than AOP when applied to OSA.

6.3.2 Integration of Services

The integration of services allows to reuse existing services and data from fields that have no links with the simulation. This allows experts to focus on their core business without having to think about how their data or services are used. Thus, it is possible to create new applications and hence new simulations by combining different services into a simulation. For example, in [Harzallah et al., 2008] the authors conducted a simulation featuring his CD++ simulator, a mapping service and a weather service to simulate a fire forest.

The integration of existing services can be done through different protocols such as web service. FRACTAL BF already presented in chapter 4 allows to connects a FRACTAL component directly to a service using several protocols (such as RMI or web service). Thus, Using FRACTAL BF enables a simulation component to connect seamlessly with a remote service. Indeed, the code encapsulated in a component only knows the local interface offered by the component that act as a proxy for accessing other components. Thus, the component model (here FRACTAL associated with FRACTAL BF) manages the connections between the content of the component and the outside world. In addition, FRACTAL BF allows to transform a FRACTAL component into a remote service. This can help in building mashup between different simulations from different simulators.

6.4 Conclusion

The integration of the JAMES II plugin system into OSA components allows to reuse all plugins from JAMES II. This includes the simulation engine, but also queues or random number generators. By reusing plug-ins from JAMES II in OSA, it now gets easier to create simulation schedulers and models for the latter because we have access to a lot of component encapsulating modeling facilities such as queues, engines, and models. Elements of these can be exchanged easily, as the reuse here means that we automatically have a strict separation of concerns that can use alternatives. Consequently the advantages of having multiple algorithms in JAMES II hold true for OSA as well. We also demonstrated that it is possible to execute a model from JAMES II in OSA. It is thus possible to use the FRACTAL ADL language to describe a simulation in OSA by using models from JAMES II.

We also worked on the integration of OSA as a backend for the experimental planning of JAMES II. The conclusions on the work needed on JAMES II to allow the integration of third-party simulators are not reported in this thesis. From the OSA side, the use of OSA as a backend is done by simply passing parameters on the command line.

Finally, we have provided solutions for achieving mashups and emulations. Mashups are easily achievable using FRACTAL BF. Using FRACTAL BF and mashups can also allow simulations to interoperate. The emulation can be done through the encapsulation of existing code into components and the interception of calls to the underlying system are done using AOP.

Chapter 7

Application and Performances

Contents

This chapter illustrates the use of techniques of reuse, separation of concerns and deployment presented in this thesis through a use case study derived from an actual study performed in our research team. Section 7.1 presents the context and the system (data storage on a peer to peer overlay network) we want to study. Section 7.2 describes how the techniques presented in this thesis serve our case study.

7.1 Use case study

We wish to study the impact of various parameters on the backup of data on a peer-to-peer storage system. Traditional backup solutions, e.g. data centers and high-end NAS appliances are highly reliable, but also tend to be very expensive. Peer-to-peer systems are an interesting alternative to obtain a storage solution with high reliability at low cost. Indeed, companies with a lot of resources distributed in different geographical locations can use them instead of expensive backup system. Similarly, small structures and individual can come together and to contribute their resources to benefit from a system data backup. Many systems have been proposed, e.g., Intermemory, CFS, Farsite, OceanStore, PAST, Glacier, TotalRecall, or Carbonite [Chun et al., 2006].

The key concept of peer-to-peer storage systems is to introduce redundancy to the data and distribute it among peers in the network. The protocol we have chosen to model for maintaining the level of redundancy is based on "erasure codes" [Luby et al., 1997, Weatherspoon and Kubiatowicz, 2002], such as Reed Solomon and Tornado. When using Erasure Codes, the original user data (e.g. files, raw data, etc.) is cut into data-blocks that are in turn divided into s initial *fragments* (or pieces) of equal size. The encoding scheme produces $s + r$ fragments that can tolerate r failures (see Figure 7.1). In other words, the original data-block can be recovered from any s of the $s+r$ encoded fragments. In a peer-to-peer storage system, these fragments are then placed on $s + r$ different peers of the network.

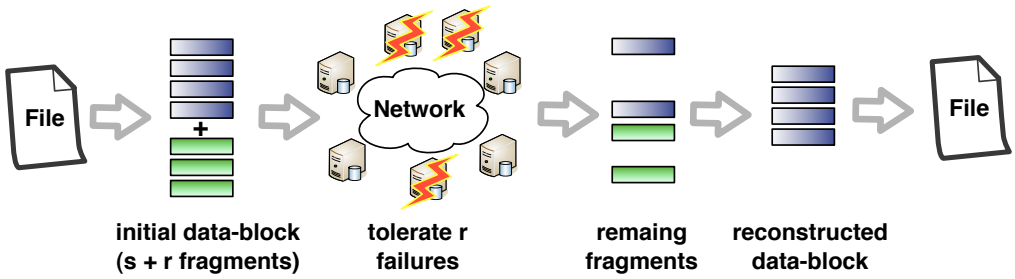

Figure 7.1: Files or raw data are cut into data-blocks. Each data-block is divided into s initial fragments, to which r fragments of redundancy are added. Any s fragments among s+r are sufficient to recover the original data-block.

A peer-to-peer systems can be full-distributed or hybrid. A hybrid system possesses a centralized server that will manage the overall meta-information of the system while a full-distributed peer-to-peer system distribute the meta-information on peers. The meta-information is the data that allow to find a piece of data in the peer-to-peer system and

know the level of redundancy. This leads to model a system where all nodes are potentially connected to all nodes, which is to have N * (N-1) connection.

PeerSim is a simulator (peersim.sourceforge.net/) that has been used for some years to evaluate P2P protocols (e.g. DHT, gossip, etc.). It has a simple cycle-based architecture that allows fast prototyping and evaluation of P2P algorithms. But PeerSim is not distributable and it seems impossible to fit in memory on a single computer the modeling of a large scale peer-to-peer system, i.e. in million of node with a large amount of references and meta-informations.

Designing such a system raises fundamental questions: How much resource (bandwidth and storage space) is necessary to maintain this redundancy and to ensure a given level of reliability? How to choose the basic set of parameters, such as s, r, r_0, to obtain an efficient utilisation of bandwidth? What is the probability that a particular system configuration results in a data loss over a given time period? How the placement strategy impacts the system behavior?

7.2 Applying Reusing Techniques Through OSA

7.2.1 Conceptual model

We choose to represent the system described above in a hierarchical fashion, with a centralized server to store meta information.

Since all peers can communicate, they must all be connected to each other. This implies N * (N-1) connections. To decrease the number of connections, we can use a component to route communications. This results in 2N connections to connect all peers among them. Figure 7.2 depicts conceptually the system under study. We can clearly see from this figure, the phenomenon of hyper-spaghetti due to too many connections between components.

Figure 7.3 shows the same conceptual model, but this time we use a shared component [Dalle et al., 2008] "network " between all peers. We see that the component "network" is defined in the component "server" and peers share this instance (shown in gray). Peers communicate through the single component instance "network" present in each peer. This results in a better readability and understandability of the model, but also the configuration files are simplified and reduced, and thus the instantiation time is reduced.

7.2.2 Implementations

To implement the previous model in OSA, we first create a new project called "P2P". Figure 7.4 shows the architecture of the "P2P" project and a part of the architecture of the "OSA" project. Each directory is a Maven project. These Maven projects are versioned and stored in a network repository to be easily reused.

Listing 7.1 shows the FRACTAL ADL definition of a “peer”. This composite component contains several components whose definitions are described in other files. In addition, the propagation of parameters allows to reuse this component without modification. The component “peer” reuse components, and is reusable.

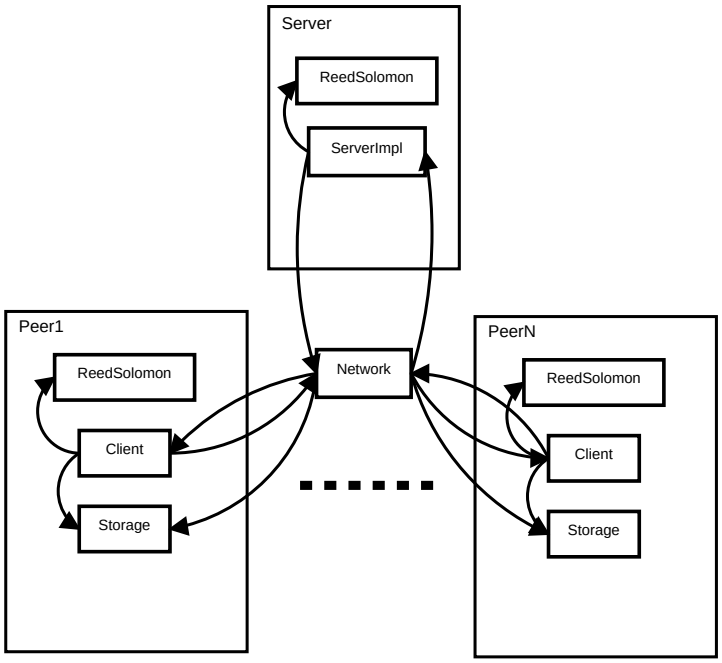

Figure 7.2: Conceptual view of the data storage model using one Network component and the hyper-spaghetti phenomenon.

This "peer" component will be reused in a more complex model, containing many "peer" components and one "server" component as shown in figure 7.3.

Listing 7.2 is the FRACTAL ADL definition of the conceptual model described by the figure 7.3. This definition reuses the "peer" definition 1000 times which represent an overall number of 5000 fractal primitive components. Inside peers component, we redefine the network component to use the one defined into the "server" component.

We realize that for large-scale architecture, FRACTAL ADL files could be large. Therefore, time required to parse all the definitions can be very long, and construction in memory impossible. The loop mechanism was not present in Fractal, we use the template-factory pattern presented in this thesis. Figure 7.5 shows the conceptual model on the use of the template-factory pattern. There is only one component "peer" which is converted into a template and one component "duplicator" which will duplicate the template to which it is connected to a number of times. Listing 7.3 shows the FRACTAL ADL definition of the use of the template-factory pattern. We add to the "peer" component and each subcomponent a template-controller to describe the way to duplicate them. Then, the component "duplicator" is parameterized and connected to the "peer" template-component. This definition is fully parameterized and could be reused easily.

The description above is part of the modeling layer. Listing 7.4 is part of the scenario layer. The "user" component play the role of a user adding data into the system. The component "peerFailure" is in charge of triggering failures on a peer, for example by cutting the connection between the component "storage" and the component "network".

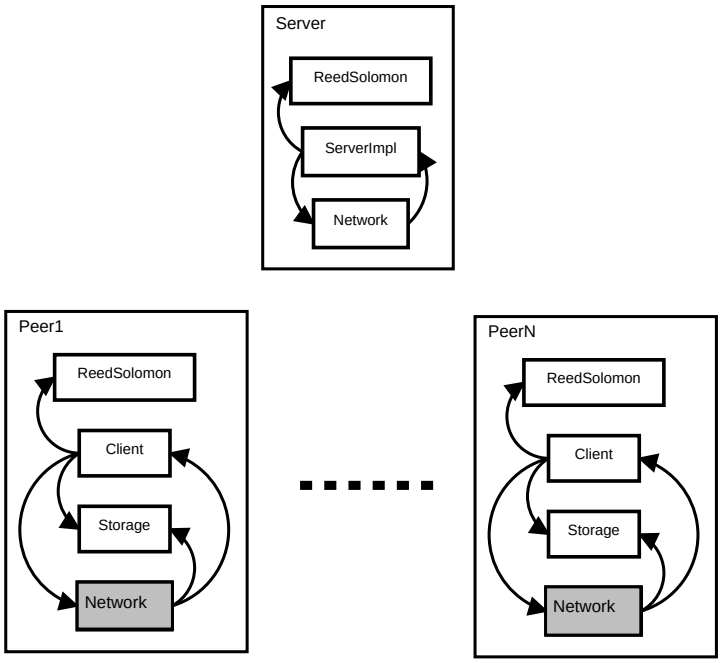

Figure 7.3: Conceptual view of the data storage model using a shared Network component (shared are in gray).

This mimics the fact that the disk of a peer is no more accessible. This definition is independent from the "peer" definition described above, but assumes the existence of a component called "peer" containing several sub-component. Indeed, the scenario involves the connection of the "user" component with the "appliImpl" component. This listing also allows to distinguish the presence of exogenous event (<exoevents>). Thus, the method "home" of the "User" component will be executed at simulation time "10" and the method "nextFailure" of the component "peerFailure" will executed at the simulation time "20".

After the modeling layer and the scenario layer, let's talk about the simulation layer represented by listing 7.5. The role of this simulation layer is to configure the control of the simulation. Here we adopt a single scheduler (superscheduler) responsible of the management of every simulation's component. Thus, the scheduler component is shared between all the composite components and is connected to all the primitive simulation's component. The only instance of the scheduler is in the component "server" and is shared in other components to avoid the phenomenon of hyper-spaghetti.

To these three layers, we can possibly add other layers. For example a layer of instrumentation, necessary to know what is going on inside the model. We want to observe disks failures and compute statistics about the failure rate of disks inside the peer to peer model.

Listing 7.6 presents the aspectj code to acquire knowledge each time a disk fail. Line 01 defines the aspect class. Lines 02-04 specify that before the execution of the peerFailure method from the PeerFailure class, we push to the StorersCollector the simulation's time

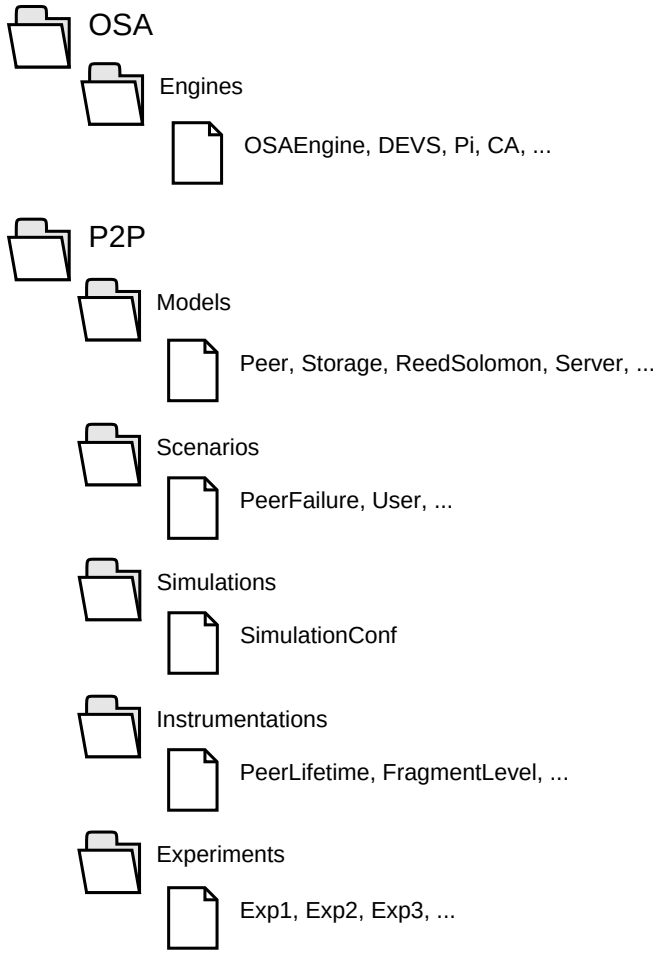

Figure 7.4: Simplified view of the architecture of the simulation project.

and the failure's id. As mentioned in section **??**, the statistical computation is managed by COSMOS but could also be done by others tools. For more detail on the COSMOS framework and how we used it through OSIF, see section 5.

7.2.3 Execution

The layers defined above are stored in a repository accessible via the Internet. Using Maven can finely tune dependencies between layers and component versions. Thus, our execution layer will define a set of dependencies (see Listing 7.7) and merge the different layers (see Listing 7.8).

Listing 7.7 shows a reduced view of the Maven configuration file (pom.xml). This file is used here to define a new "experiment" Maven project called "exp-1000peers". The first part of the file defines the parent project, then identifies the project by groupId, parentId and version number. We deliberately hide the use of various plugin that serve to configure the project because it is verbose. These are among other plugin to configure the compila-

Listing 7.1: FRACTAL ADL definition of a peer component.

```
<?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="p2p.models.peer" arguments="storersize,s,r">

        <component name="reedsolomon" definition="p2p.models.ReedSolomon(s=>${s},r
            =>${r})" />

        <component name="appliImpl" definition="p2p.models.PeerImpl" />

        <component name="storer" definition="p2p.models.Storer(hddsize=>${
            storersize})">
                <exoevents signature="storer">
                        <exoevent name="register" type="StartOfCall" time="0"
                            method="boot" />
                </exoevents>
        </component>

        <component name="network" definition="p2p.models.Network" />

        <binding client="appliImpl.storers" server="storer.storer" />
        <binding client="appliImpl.server" server="network.network" />
        <binding client="appliImpl.reedsolomon" server="reedsolomon.reedsolomon" />
        <binding client="network.peer" server="storer.storer" />

</definition>
```

tion and execution of this project. Then, dependencies are defined. There are dependencies to sub-project “models”, “scenarios”, “instrumentations” and “simulations”. Thanks to transitivity, the dependencies of the project “p2p.experiments.exp-1000peers” benefits from dependencies of projects it depends. Dependencies of the sub-project “p2p.simulations” to the project “OSA.OSAengines” are de facto added to the dependencies of the project “p2p.experiments.exp-1000peers”.

Listing 7.8 represents the experiment layer which will be stored in the Maven project described above. We define the different layers that we want to merge and we set the parameters to define a simulation application. Fixed and stored in a Maven repository with all the dependencies, replayability is assured as long as the Maven repository exists.

In a future study, we may want to start again from previous experience and to vary some parameter or to add elements to the simulation, such as adding an new scenario. Listing 7.9 shows how to extend the previous experiment and change just one parameter in order to have 2000 peers instead of 1000, and add in addition to the scenario “p2pscenario”, a second layer of scenario “p2pscenario2”.

7.2.4 Deployement

The deployment of the model written in Listing 7.2 is described below in Listing 7.10. The component “server” will be instantiated locally while the first 250 peers will be instantiated on the computational resources associated with the virtual node “cr1”, the following 250 peers will be instantiated on the computational resources associated with the virtual node

Listing 7.2: FRACTAL ADL definition of the P2P model.

```
<?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="p2p.models.p2pmodel1000" arguments="hddsize,s,r,r0,theta">

        <component name="server">
                <component name="serverImpl" definition="p2p.models.ServerImpl(${r0
                    },${theta},${hddsize})" />
                <component name="network" definition="p2p.models.Network" />
                <component name="reedsolomon" definition="p2p.models.ReedSolomon(${
                    s},${r})">
                <binding client="serverImpl.reedsolomon" server="reedsolomon.
                    reedsolomon" />
                <binding client="serverImpl.peers" server="network.network" />
                <binding client="network.server" server="serverImpl.server" />
        </component>

        <component name="peer0" definition="p2p.models.peer(${hddsize},${s},${r})">
                <component name="network" definition="server/network" />
        </component>

        <component name="peer1" definition="p2p.models.peer(${hddsize},${s},${r})">
                <component name="network" definition="server/network" />
        </component>

        [...]

        <component name="peer999" definition="p2p.models.peer(${hddsize},${s},${r})
            ">
                <component name="network" definition="server/network" />
        </component>

</definition>
```

Listing 7.3: FRACTAL ADL definition of the P2P model using template-factory pattern.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath:
    //fr/inria/osa/simulationAPIs/basicAPI/adl/stdsim.dtd">
<definition name="p2p.models.p2pmodel" arguments="hddsize,s,r,r0,theta,loop">


        <component name="server">
                <component name="serverImpl" definition="p2p.models.ServerImpl(${r0
                    },${theta})" />
                <component name="network" definition="p2p.models.Network" />
                <component name="reedsolomon" definition="p2p.models.ReedSolomon(${
                    s},${r})">
                <binding client="serverImpl.reedsolomon" server="reedsolomon.
                    reedsolomon" />
                <binding client="serverImpl.peers" server="network.network" />
                <binding client="network.server" server="serverImpl.server" />
        </component>

        <component name="peer" definition="p2p.models.peer(${hddsize},${s},${r})">
                <component name="network" definition="server/network">
                        <template-controller desc="sharedPrimitiveTemplate"/>
                </component>
                <component name="storer">
                        <template-controller desc="simPrimitiveTemplate"/>
                </component>
                <component name="appliImpl">
                        <template-controller desc="simPrimitiveTemplate"/>
                </component>
                <component name="reedsolomon">
                        <template-controller desc="simPrimitiveTemplate"/>
                </component>
                <template-controller desc="simCompositeTemplate"/>
        </component>

        <component name="DuplicatorFactory" definition="OSA.misc.Duplicator(
            iteration=>${loop})" />

        <binding client="DuplicatorFactory.template" server="peer.component" />

</definition>
```

Listing 7.4: FRACTAL ADL definition of a scenario for the P2P model.

```
<?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="p2p.scenario.p2pscenario" arguments="fileperuser,alpha,zeta">
        <component name="peer">
                <component name="user" definition="p2p.scenario.User(${fileperuser
                    })">
                    <exoevents signature="user">
                        <exoevent name="user" type="StartOfCall" time="10"
                            method="start" />
                    </exoevents>
                </component>
                <component name="peerfailure" definition="p2p.scenario.PeerFailure(
                    alpha=>${alpha},zeta=>${zeta})">
                    <exoevents signature="storer">
                        <exoevent name="start" type="StartOfCall" time="20"
                            method="nextFailure" />
                    </exoevents>
                </component>

                <binding client="user.client" server="appliImpl.client" />
                [...]
                [...]

        </component>
</definition>
```

Listing 7.5: FRACTAL ADL definition of a simulation control for the P2P model.

```
<?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="p2p.simulation.p2psimulation">

        <component name="server">
                <component name="superscheduler" definition="OSA.Engines.OSAEngine.
                    SuperScheduler" />
                <binding client="serverImpl.superscheduler" server="superscheduler.
                    superscheduler" />
        </component>

        <component name="peer">
                <component name="superscheduler" definition="./superscheduler" />
                <binding client="peerfailure.superscheduler" server="superscheduler
                    .superscheduler" />
                <binding client="user.superscheduler" server="superscheduler.
                    superscheduler" />
                <binding client="appliImpl.superscheduler" server="superscheduler.
                    superscheduler" />
                <binding client="storer.superscheduler" server="superscheduler.
                    superscheduler" />
        </component>

</definition>
```

Listing 7.6: AspectJ code to acquire knowledge each time a disk fail.

```
01 public aspect diskFailure {
02   before(PeerFailure failure): execution(void PeerFailure.peerFailure()) && this
    (failure) {
03     DiskFailureCollector.pushFromAspect(failure.simulationController_.
    getSimulationTime(), failure.failureId_);
04   }
05 }
```

Listing 7.7: Maven configuration file of the "exp-1000peers" experiment project.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
            org/maven-v4_0_0.xsd">
        <parent>
                <artifactId>p2p</artifactId>
                <groupId>p2p</groupId>
                <version>1.0</version>
        </parent>
        <modelVersion>4.0.0</modelVersion>
        <groupId>p2p.experiments</groupId>
        <name>1000 peers</name>
        <version>1.0</version>
        <artifactId>exp-1000peers</artifactId>

        [...]

        <dependencies>
                <dependency>
                        <groupId>p2p</groupId>
                        <artifactId>models</artifactId>
                        <version>1.0</version>
                </dependency>
                <dependency>
                        <groupId>p2p</groupId>
                        <artifactId>scenarios</artifactId>
                        <version>1.0</version>
                </dependency>
                <dependency>
                        <groupId>p2p</groupId>
                        <artifactId>instrumentations</artifactId>
                        <version>1.0</version>
                </dependency>
                <dependency>
                        <groupId>p2p</groupId>
                        <artifactId>simulations</artifactId>
                        <version>1.0</version>
                </dependency>

                [...]

        </dependency>
</project>
```

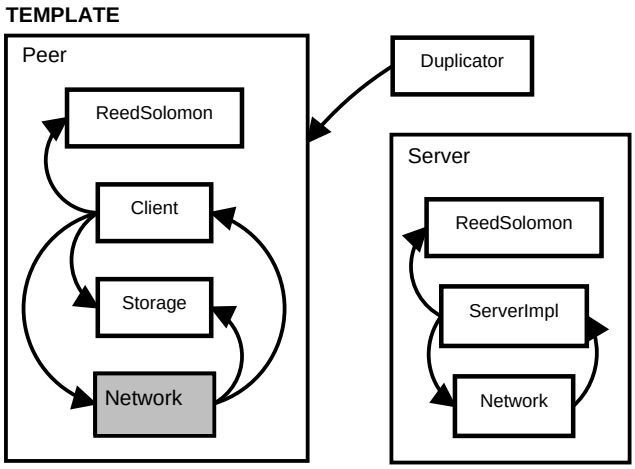

Figure 7.5: Conceptual view of the P2P model using the template-factory pattern.

Listing 7.8: FRACTAL ADL definition of an experiment for the P2P model

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

<definition name="p2p.experiments.exp-1000peers"
        extends="p2p.models.p2pmodel(s=>5,r=>5,r0=>9,theta=>12,loop=>1000),
        p2p.scenarios.p2pscenario(fileperuser=>100,alpha=>17520,zeta=>72),
        p2p.simulations.p2psimulation,
        p2p.instrumentations.p2pinstrumentation">
</definition>
```

"cr2", and so on. During the execution of the simulation is associated to each virtual node a physical resource on which the fractal factory can connect and deploy the component. The use of the pattern template-factory described above (see Listing 7.3) prevents the use of FractalRMI on new duplicated components as they don't exists at the Fractal loader factory level. The components are duplicated at runtime by the "duplicator" component. However, it is possible to create a component "duplicator" who would be in charge to deploy the components in the manner of the Fractal loader factory. No other changes are required on the code to deploy and connect remote components. Indeed, the code inside the component only communicates with the internal interface of the component, and the Fractal component model handles the remote communication between the external interfaces of components.

Another solution for application deployment is the use of fractal Fractal-BF. Fractal-BF transforms a component into a component providing a service interface or connect a Fractal component to a service using various protocols (RMI, WebService, OSGi, Rest, ...). Figure 7.6 describes the development of a simulation as the overall composition of two simulations. The first simulation on the "computer1" contains several "peer" (use of template-factory) and a "server". The second simulation on the "computer2" contains

Listing 7.9: FRACTAL ADL definition of another experiment for the P2P model

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

<definition name="p2p.experiments.exp-2000peers"
        extends="p2p.experiments.exp-1000peers,
        p2p.models.p2pmodel(loop=>2000),
        p2p.scenarios.p2pscenario2(gamma=>42),">
</definition>
```

Listing 7.10: FRACTAL ADL definition of a deployment for the P2P model using FractalRMI

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath:
    //fr/inria/osa/simulationAPIs/basicAPI/adl/stdsim.dtd">
<definition name="p2p.deployments.p2pdeploy">

        <component name="server">
        </component>

        <component name="peer0">
                <virtual-node name="cr1"/>
        </component>
        [...]
        <component name="peer249">
                <virtual-node name="cr1"/>
        </component>

        <component name="peer250">
                <virtual-node name="cr2"/>
        </component>
        [...]
        <component name="peer449">
                <virtual-node name="cr2"/>
        </component>

        <component name="peer500">
                <virtual-node name="cr3"/>
        </component>
        [...]
        <component name="peer749">
                <virtual-node name="cr3"/>
        </component>

        <component name="peer750">
                <virtual-node name="cr4"/>
        </component>
        [...]
        <component name="peer999">
                <virtual-node name="cr4"/>
        </component>


</definition>
```

several "peer" but no "server". Indeed, this second simulation must integrate with the first simulation to make a single simulation containing twice as many peers. Any peer can communicate together and with the server, we replace the implementation of the component "network" with another implementation which will have the role to route efficiently any communication between peers located indifferently on the computer "computer1" and "computer2".

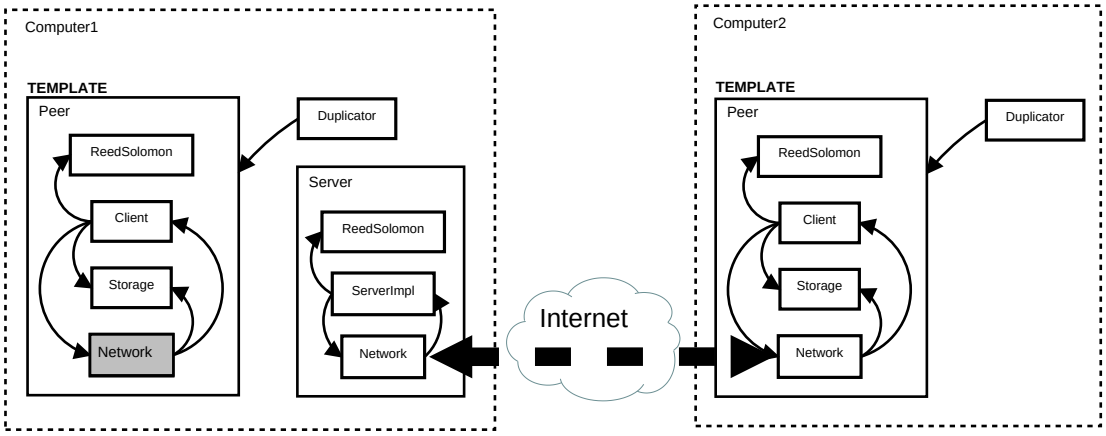

Figure 7.6: Conceptual model of a distributed simulation of the P2P model.

Listing 7.11 shows the deployment layer that will replace the implementation of the component "network" located in the component "server" and initiates RMI connections using Fractal-BF. The <exporter> turn the "network" interface of the "network" component into a service using the RMI protocol. The <binder> try to establish a connection to a remote service called "network" trough the RMI protocols. Parameterizing this definition allows to reuse them further, by specifying address and port of the local and remote computer.

7.3 Performances

In this section, we present results we have obtained through various simulation experiments. The simulated system is the data storage system running on a P2P overlay network introduced above. We show that this model scales well and present a quantitative assessment in terms of reuse, instrumentation and integration.

The challenges that we face are creating simulations of several million components, distributed over several computational node, in order to reduce the startup time of the simulation. In addition, separation of concerns should promote the reuse of various concerns of the simulation, such as model, scenario, instrumentation, and deployment.

Section 7.3.1 shows the contribution of our work in terms of performance in FRACTAL application. Section 7.3.2 shows comparative studies of the starting time of simulation based on the number of node and peers.

Listing 7.11: FRACTAL ADL definition of a deployment for the P2P model using FractaBF.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN" "classpath:
    //fr/inria/osa/simulationAPIs/basicAPI/adl/stdsim.dtd">
<definition
        name="p2p.deployments.p2pdeploybf"
        arguments="remoteaddress,remoteport,localaddress,localport">


        <component name="network" definition="p2p.deployments.ProxyNetwork(address
            =>${address},port=>${port})" />

        <exporter type="rmi" interface="network.network">
                <parameter name="serviceName" value="network" />
                <parameter name="hostAddress" value="${localaddress}" />
                <parameter name="port" value="${localport}" />
        </exporter>

        <binder type="rmi" interface="network.remotenetwork">
                <parameter name="serviceName" value="network" />
                <parameter name="hostAddress" value="${remoteaddress}" />
                <parameter name="port" value="${remoteport}" />
        </binder>

</definition>
```

7.3.1 FRACTAL Performance

In this section, we provide a quantitative answer about the overhead of using FRACTAL, and about the improvement we are providing. The FRACTAL component model has a cost in comparison to a solution in full Java. A simple method call between two components takes fifty times more time than a method call between two Java classes. To assess this, we developed a very simple application consisting of 2 classes. The first class contains a method that makes a given number of times a method call to a method of the second class. The method of the second class does nothing so that we can measure the time spent in method calls. The tests have been run on a 2.2Ghz dual core machine with 2GB of RAM and Java 1.6.0_07. The full Java version of this application run 10^{10} method calls in 15 seconds, while the FRACTAL version (using FRACTAL2.5.3-Snapshot) of the same application run 10^8 method calls in 9 seconds. There is a speed factor between the two of 50. In view of the benefits provided by the FRACTAL component model we consider that this factor is acceptable. Indeed, this cost is relativized by the fact that in most applications, the calculations within one component takes more time than the cost of an external method call to another component. This is due to the fact that components are bigger than objects. A component is a unit of modeling and may contain multiple objects which is a unit of coding. The speed factor drawback is only for communication between components, not for communication between objects inside a component. It is also relativized by the fact that FRACTAL supports the means for providing communication between components, which enables faster development of application and avoids many bugs.

In our work related to the performance of very large scale simulation, we first wanted to be able to run on a single machine as many components as possible. Therefore, we worked to improve the performance of FRACTAL, by optimizing the data structure that was inadequate for a very large architecture. Then, we studied the performance of the FRACTAL templates described in section 4.5.1.

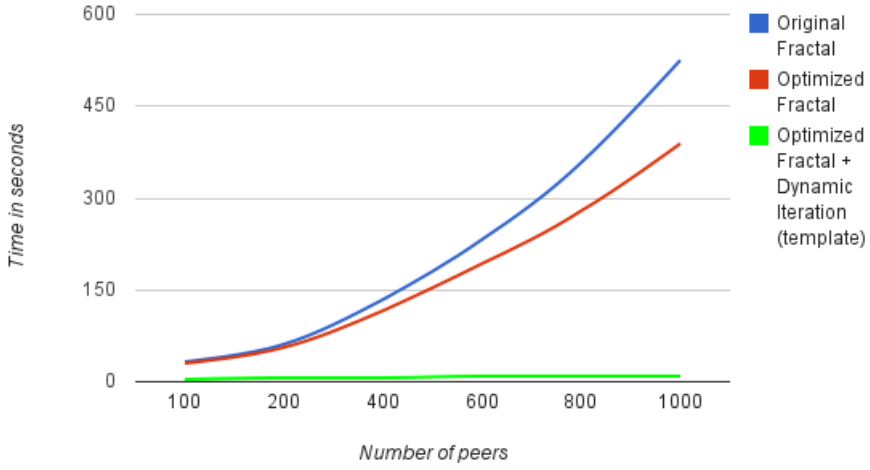

Figure 7.7: Time to start simulations when varying the number of peers in the simulation.

Figure 7.7 represents the time needed to start the simulation depending on the number of simulated peers. The starting time of the simulation includes the instantiation of components, their initialization and the initialization of the simulation. The upper (blue) curve was obtained using the base implementation of FRACTAL. The middle(red) curve was obtained with the modifications made to the data structure of FRACTAL. The lower (green) curve was obtained with the use of the dynamic architecture based on templates. This figure shows that the optimizations on the data structures are increasingly important as the number of peers increases. But this figure is used mainly to see that the use of the dynamic architecture is a significant performance improvement of the startup time of large simulations.

7.3.2 Deployment Performance

We saw in section 4.4 that it was possible, thanks to FRACTAL, FRACTAL BF or FRACTAL RMI to deploy a distributed application seamlessly. By deploying our P2P application with FRACTAL BF, we obtain the results shown in figures 7.8 and 7.9. These results were obtained after measuring the time to start a simulation on a homogeneous set of machines on the Grid'5000 grid. Grid'5000 is a scientific instrument for the study of large scale parallel and distributed systems. It aims at providing a highly reconfigurable,

controlable and monitorable experimental platform to its users with thousands of nodes that can be exclusively alligned for experiments.

Figure 7.8 shows the starting time of a simulation based on the number of computational node used. The lower (blue) curve represents 100 peers per node, the middle (red) curve represents 1000 peers per node, and the upper (yellow) curve represents 10000 peers per node. The yellow curve shows that multiplying the number of peers by 50, it does not increase the startup time of the simulation by 50, but a little more than 2. However, the start-up time is not constant because the Server component is the bottleneck of the application and it must manage more connections as the number of peers increase. The scheduler also has more events to store in the waiting list.

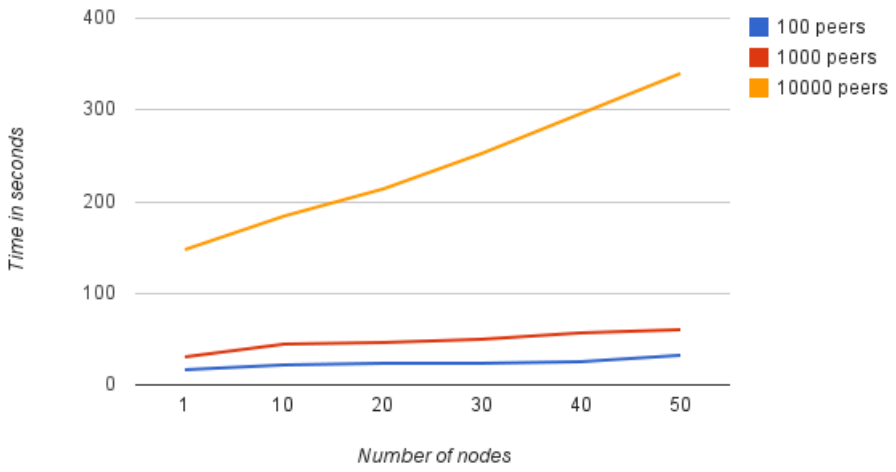

Figure 7.8: Evolution of the startup time depending on the number of nodes (with a fixed number of peers per node).

Figure 7.9 shows the starting time of a simulation depending on the number of node on which it is distributed. It is expected that the time decreases with the number of nodes used in parallel. Whether the lower (blue) curves (simulation of 10000 peers), the middle (red) curves (simulation of 100000 peers) or the upper (yellow) curves (simulation of 500000 peers), the time decreases as the number of nodes increases. However, we see a threshold phenomenon and even the blue curve increases at the end. The startup time of the simulation does not decreases proportionally with the number of nodes involved in the distribution, because the model contains an important bottleneck with the presence of the Server component that receives all communications from peers. The number of peers who want to register to the Server component does not change depending on the distribution, but instead of having a process that would establish a sequential connection with the server, there are as many connection as there are nodes. Managing conflicts between

different connections consumes resources. This means that distributing a simulation with a too small number of peers on each computational node is not efficient. We must find a balance between distribution and computation. For example, we obtained a nearly perfect distribution in the following case where the distribution divides the startup time by the number of node: 20000 peers takes on 1 node 354 seconds, 170seconds on 2 nodes, and 95 seconds on 4 nodes. Moreover, we see that the yellow curve shows no value for 1 and 10 nodes. This is because it is simply impossible to run a simulation of this size without exceeding the memory capacity of the machines.

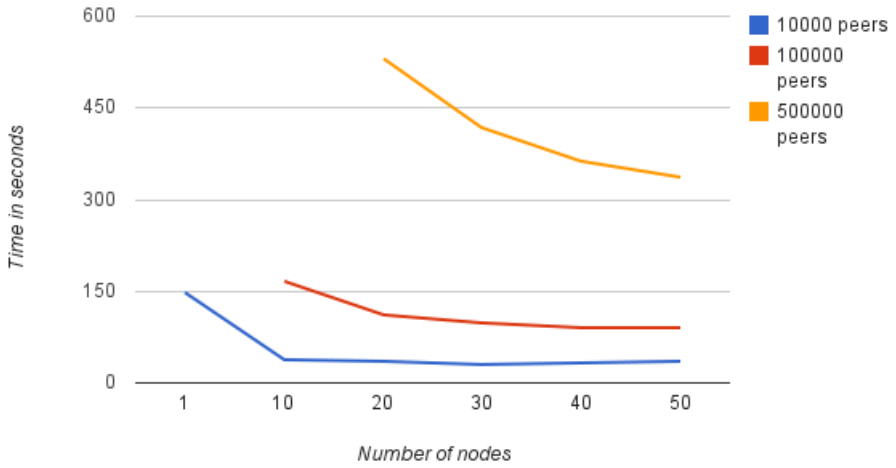

Figure 7.9: Time to start a distributed simulation by the number of computational nodes involved.

As we mentioned above, the use of multiple computational resources may be required to run large models. Distribution of this model on different resources requires an optimization work that must be done by an expert. Through our independent layer approach, an expert can work on the deployment of the various components regardless of the development process.

7.4 Conclusion

Although FRACTAL ADL can already improve reusability through inheritance and overloading that divides architecture into several definitions, the changes we have made on the template mechanism allow to go one step further, allowing to transform components of an existing architecture into clonable components, without altering the original architecture while improving reusability.

	without dist.	with dist.
without opt.	1000	N/A
with opt.	30000	>500000

Figure 7.10: Maximum number of peers instantiated.

Table 7.10 summarises our results. The number of peers that we were able to instantiate on a single computational node without our optimizations was 1000. With our optimizations, that number goes up to 30000, a size factor of 30. In terms of speed, the instantiation of the 1000 peers took 534 seconds. With our optimizations the same instantiation takes only 9 seconds, a speed factor of 60. If we add the distribution, the number of peers can go up to more than 500000 peers, for a time of 330 seconds. Although it is possible to continue to increase the size of the architecture and the number of node used, the goal are now to improve the parallelization of the simulation engine in order to benefit from parallel execution.

Our initial challenge was to be able to run simulations with millions of Fractal components. This challenge has been successfully met, since in our case study, a peer has 7 primitive components, which corresponds to 3.5 million components for a simulation of 500000 peers excluding shared components and composite components.

Chapter 8

Conclusion

Contents

In the introduction to this thesis, we posed the problem of reusing and integrating elements of existing simulations. Indeed, reuse increases dependability, is less error prone, makes better use of complementary expertises, improves standards compliance, and accelerates development. These benefits can be observed in the development of simulation software, at all levels of the architecture (modeling, scenario, instrumentation, simulation, deployment , experimentation, and so on).

We started by discussing the design of simulation tools, how it is important to pay particular attention, and how the reuse could be beneficial. Then we surveyed the state of the art techniques for reuse found in current simulation software products and we reach the conclusion that these software does not allow to take advantage of all the benefits of reuse and it was interesting to look at alternatives for reuse. We propose a proof of concept with the realization of a prototype named OSA. This platform uses the principles of component-oriented programming, aspect-oriented programming, and an architecture description language that is flexible and offers advanced composition mechanisms such as multiple inheritance and overloading. Then we describe a practical solution which exploits such software engineering tools in the development of reusable model, scenario and simulation engine, and in the configuration of the distributes execution seamlessly. We also proposed OSIF, a tool to instrument and analyze applications such as simulation, based on the same techniques and software engineering tools. OSIF allows a complete separation of concerns between modeling, instrumentation and data processing; we showed that it favors the validation of results by allowing the sharing of analysis between the real system and the simulated system; we also showed that it allows to manage and optimize the flow of simulation data whether we want to analyse data during execution or post-mortem; and we showed that it allows to design and compose complex instrumentations and data processings in a simple way. Then we showed how the use of these software engineering tools allows the integration of existing simulation elements. Finally, we validate the performance of our approach with a study of performance in terms of reuse, execution time, and scalability of a P2P simulation. If it is difficult to measure quantitatively the benefits of reuse, the work conducted on the advanced features of FRACTAL allowed us to cross a gap in architecture size and in the starting time of simulation.

8.1 Contributions

We have shown in this thesis that the use of software engineering techniques allowed to take advantage of reuse in two main areas which are:

- the production of tools that promote reuse through the use of component programming and the layered approach, but it can also go through the use of other techniques such as the plug'n simulate mecanism from JAMES II.
- the integration of third-party tools not intended to be reusable, through the use of aspect-oriented programming.

We validate our approach with the realization of two software contributions: OSA and OSIF (available on http://osa.gforge.inria.fr). Despite the fact that OSA has been pre-

sented by Olivier Dalle in [Dalle, 2006, Dalle, 2007a], OSA is one of the contribution of this thesis. First, due to the major development to come from the OSA concept to the OSA project. Second, due to the conceptual work done on OSA to enhance the original concept.

OSA OSA is a collaborative *platform* for component-based discrete-event simulation. OSA is based on the FRACTAL component model which allows to separate the functional and nonfunctional concerns (controllers in the membrane manages the non-functional aspect of component). FRACTAL ADL allows separating functional concerns into different layers and provides multiple inheritance, overloading and dynamic architecture. Using the overloading technique of FRACTAL ADL, combined with the use of aspect-oriented programming, OSA allows to build advanced scenarios that result in the structural change of a model, but without requiring the changes to be applied to the original model itself. The use of FRACTAL RMI and FRACTAL BF allows to distribute an application over several computing nodes, which achieves a significant size architecture it would be unattainable on a single compute node. The management of remote communications by FRACTAL also allows mashup with remote services. The integration of existing code into components allows the integration of simulation elements as the elements of the JAMES II framework. The emulation can be done through the encapsulation of existing code into components and the interception of calls to the underlying system are done using AOP.

OSIF OSIF is a base for creating, analysing and validating instrumentations. Benefits of OSIF are multiple: (i) OSIF allows a complete separation of concerns between modeling, instrumentation and data processors; (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 processors in a simple way. OSIF has been designed so that there are no connection between the simulator and instrumentation and data processors. 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.

8.2 Perspectives

OSA OSA is an experimental platform, set up to validate our approach on separation of concerns and techniques for reuse. In order to federate a user community and allow the sharing of OSA components, we must work on a more high-level architecture that helps the user with the simulation approach. This requires the development of automation tools and graphical interface, as well as tools to guide the user. Of course, with the multitude of tools available to support simulation, our research is about the integration of these different tools in OSA rather than to developing new tools dedicated to OSA.

OSIF 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 processors 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 and al. explained in [Himmelspach et al., 2008]. 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 strip out the beginning of a simulation or refining inputs to obtain the best inputs combination for a study.

Interoperability The work done in this thesis provides solutions to known problems, such as interoperability between simulations. Indeed, in distributed simulations, interoperability between distributed components is essential to ensure a consistant global behavior. All stakeholders must communicate and interact in a distributed way by following a common framework which is set by an architecture of distributed simulation. HLA allows to create a global simulation composed of interacting distributed simulations without being rewritten. Using HLA, simulations can communicate with other simulations regardless of underlying platforms. However, the HLA model has limitations[Davis and Moeller, 2002] and does not support real-time information. An original approach to composition could be based on Web services using the RESTful communication protocol such as the one described in [Al-Zoubi and Wainer, 2009]. This web-based protocol offers through a set of simple communication primitives, all the necessary means for the composition of distributed simulations. Research efforts should now focus on integration and interoperability without modification of existing models as well as taking into account information in real time in order to use simulation as tools for decision making.

Workflow Another interesting research perspective is to follow the evolution of works that address issues of simulations replayability such as [Perrone et al., 2009]. Our experimentation platform is a platform that we can consider today as a low-level platform. To ensure a minimal traceability and ensure simulation replayability, we use the versioning and backup of simulation projects into Maven repositories. An interesting research perspective is the addition of a layer of control that guides the user through these choices to promote good practices of the simulation.

Bibliography

[osg, 2004] (2004). *OSGi Technical Whitepaper*. OSGi Alliance. Revision 3.0. `www.osgi.org`. 13

[Agresti and Evanco, 1992] Agresti, W. and Evanco, W. (1992). Projecting software defects from analyzing Ada designs. *Software Engineering, IEEE Transactions on*, 18(11):988–997. 10

[Al-Zoubi and Wainer, 2009] Al-Zoubi, K. and Wainer, G. A. (2009). Using rest web-services architecture for distributed simulation. In *PADS*, pages 114–121. IEEE Computer Society. 118

[Aldrich et al., 2002] Aldrich, J., Chambers, C., and Notkin, D. (2002). ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197. AP. 13

[Andradóttir, 1998] Andradóttir, S. (1998). *Handbook of Simulation*, chapter 9- Simulation Optimization, pages 307–334. EMP & Wiley. 26

[Atkinson et al., 2000] Atkinson, C., Bayer, J., and Muthig, D. (2000). Component-based product line development: the KobrA approach. In *Software product lines: experience and research directions: proceedings of the First Software Product Lines Conference (SPLC1), August 28-31, 2000, Denver, Colorado*, page 289. Springer Netherlands. 15

[Balci, 2003] Balci, O. (2003). Verification, validation, and certification of modeling and simulation applications. In Chick, S., Sánchez, P. J., Ferrin, D., and Morrice, D. J., editors, *Proceedings of the 2003 Winter Simulation Conference*, pages 150–158. Winter Simulation Conference. 2, 37

[Balci and Nance, 1992] Balci, O. and Nance, R. E. (1992). The simulation model development environment: an overview. In Swain, J. J., Goldsman, D., Crain, R. C., and Wilson, J. R., editors, *Proceedings of the 1992 Winter Simulation Conference*, pages 726–736, New York, NY, USA. ACM. 40

[Banks, 1999] Banks, J. (1999). Introduction to simulation. In *Proceedings of the 31st conference on Winter Simulation Conference: Simulation—a bridge to the future-Volume 1*, page 13. ACM. 17

[Banks et al., 2004] Banks, J., Carson II, J. S., Nelson, B. L., and Nicol, D. M. (2004). *Discrete-Event System Simulation*. Prentice Hall, 4th edition. 18, 26, 72

[Barnes et al., 1988] Barnes, B., Durek, T., Gaffney, J., and Pyster, A. (1988). A framework and economic foundation for software reuse. In *Software reuse: emerging technology*, pages 77–88. IEEE Computer Society Press. 10

[Barns and Bollinger, 1991] Barns, B. and Bollinger, T. (1991). Making reuse cost-effective. *Software, IEEE*, 8(1):13–24. 8

[Barr, 2004] Barr, R. (2004). *An efficient, unifying approach to simulation using virtual machines*. PhD thesis, Cornell University. 55

[Basili et al., 1990] Basili, V., Rombach, H., Bailey, J., and Delis, A. (1990). Ada reusability and measurement. *Computer Science Technical Report Series; Vol. CS-TR-2478*, page 25. 11

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional. 14

[Begg et al., 2006] Begg, L., Liu, W., Pawlikowski, S., Perera, S., and Sirisena, H. (2006). Survey of simulators of next generation networks for studying service availability and resilience. 21

[Bodoff et al., 2002] Bodoff, S., Green, D., Haase, K., Jendrock, E., Pawlan, M., and Stearns, B. (2002). *The J2EE tutorial*. 13

[Bovet et al., 2002] Bovet, D., Cesati, M., and Oram, A. (2002). *Understanding the Linux kernel*. O'Reilly & Associates, Inc. Sebastopol, CA, USA. 58

[Browne et al., 1990] Browne, J., Lee, T., and Werth, J. (1990). Experimental evaluation of a reusability-oriented parallel programming environment. *Software Engineering, IEEE Transactions on*, 16(2):111–120. 10

[Bruneton et al., 2006] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani., J. (2006). The fractal component model and its support in java. *Software Practice & Experience*, 36(11-12). Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems. 13, 24, 77, 85

[Bruneton et al., 2004] Bruneton, E., Coupaye, T., and Stefani, J. (2004). The fractal component model specification. Available from http://fractal.objectweb.org/specification/ [Last accessed: 11/30/2010]. Draft version 2.0-3. 27, 28

[Bures et al., 2006] Bures, T., Hnetynka, P., and Plasil, F. (2006). Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48. IEEE. 15

[C.A.R. Hoare, 1978] C.A.R. Hoare (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677. 58

[Card et al., 1986] Card, D., Church, V., and Agresti, W. (1986). Empirical study of software design practices. *IEEE Transactions on Software Engineering*, 12(2):264–271. 10

[Chang and Collet, 2007] Chang, H. and Collet, P. (2007). Compositional patterns of non-functional properties for contract negotiation. *Journal of Software*, 2(2):52–63. 11

[Chen and Lee, 1993] Chen, D. and Lee, P. (1993). On the study of software reuse using reusable C++ components. *Journal of Systems and Software*, 20(1):19–36. 10

[Chidamber and Kemerer, 1994] Chidamber, S. and Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on software engineering*, pages 476–493. 10

[Chun et al., 2006] Chun, B.-G., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M. F., Kubiatowicz, J., and Morris, R. (2006). Efficient replica maintenance for distributed storage systems. In *Proc. of NSDI*, pages 45–48. 96

[Clarke et al., 2001] Clarke, M., Blair, G., Coulson, G., and Parlavantzas, N. (2001). An efficient component model for the construction of adaptive middleware. In *Proceedings of Middleware'01*. 13

[Conan et al., 2007] Conan, D., Rouvoy, R., and Seinturier, L. (2007). Scalable Processing of Context Information with COSMOS. In Indulska, J. and Raymonds, K., editors, *Proc. 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, volume 4531 of *lncs*, pages 210–224, Paphos, Cyprus. Springer-Verlag. 73, 74, 77

[Dahmann et al., 1997] Dahmann, J., Fujimoto, R., and Weatherly, R. (1997). The department of defense high level architecture. In *Proceedings of the 29th conference on Winter Simulation Conference*, pages 142–149. IEEE Computer Society. 23

[Dall et al., 2010] Dall, O., Lapides, J., and Locke, T. (2010). Rapid rails with hobo. available from www.hobocentral.net/books [Last accessed March 2011]. 12

[Dalle, 2006] Dalle, O. (2006). OSA: an Open Component-based Architecture for Discrete-Event Simulation. In *20th European Conference on Modeling and Simulation (ECMS)*, pages 253–259, Bonn, Germany. 24, 64, 117

[Dalle, 2007a] Dalle, O. (2007a). 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. 24, 117

[Dalle, 2007b] Dalle, O. (2007b). The OSA Project: an xample of Component Based Software ngineering Techniques pplied to Simulation. In Vakilzadian, H., editor, *Proc. of the Summer Computer Simulation Conference (SCSC'07)*, pages 1155–1162, San Diego, CA, USA. Invited paper. 31

[Dalle and Mrabet, 2007] Dalle, O. and Mrabet, C. (2007). An instrumentation framework for component-based simulations based on the separation of concerns paradigm. In *Proc. of 6th EUROSIM Congress (EUROSIM2007)*, Ljubljana, Slovenia. 72

[Dalle et al., 2010] Dalle, O., Ribault, J., and Himmelspach, J. (2010). Design considerations for M&S software. In *Proceedings of the 2009 Winter Simulation Conference (WSC'09)*, pages 944–955. IEEE. 36

[Dalle et al., 2008] Dalle, O., Zeigler, B. P., and Wainer, G. A. (2008). Extending DEVS to support multiple occurrence in component-based simulation. In Mason, S. J., Hill, R. R., Mönch, L., and Rose, O., editors, *Proceedings of the 2008 Winter Simulation Conference*. 24, 64, 97

[Davis and Anderson, 2004] Davis, P. and Anderson, R. (2004). Improving the composability of DoD models and simulations. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, 1(1):5. 49

[Davis, 1993] Davis, T. (1993). The reuse capability model: a basis for improving an organization's reuse capability. In *Software Reusability, 1993. Proceedings Advances in Software Reuse., Selected Papers from the Second International Workshop on*, pages 126–133. IEEE. 10

[Davis and Moeller, 2002] Davis, W. and Moeller, G. (2002). The High Level Architecture: is there a better way? In *Proceedings of the 1999 Winter Simulation Conference*, volume 2, pages 1595–1601. IEEE. 118

[des Rivières and Wiegand, 2004] des Rivières, J. and Wiegand, J. (2004). Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383. 33

[Dowling and Cahill, 2001] Dowling, J. and Cahill, V. (2001). The k-component architecture meta-model for self-adaptive software. *Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88. 13

[Dunn and Knight, 1991] Dunn, M. and Knight, J. (1991). Software reuse in an industrial setting: a case study. In *Proceedings of the 13th international conference on Software engineering*, pages 329–338. IEEE Computer Society Press. 11

[Endres and Rombach, 2003] Endres, A. and Rombach, D. (2003). *A Handbook of Software and Systems Engineering*. Pearson Education Ltd., Essex, England. 36, 43

[Favaro, 1991] Favaro, J. (1991). What price reusability?: a case study. In *ACM SIGAda Ada Letters*, volume 11, pages 115–124. ACM. 10

[Fishman, 2001] Fishman, G. (2001). *Discrete-event simulation: modeling, programming, and analysis*. Springer Verlag. 18

[Frakes et al., 1991] Frakes, W., Biggerstaff, T., Prieto-Diaz, R., Matsumura, K., and Schaefer, W. (1991). Software reuse: is it delivering? In *Proceedings of the 13th international conference on Software engineering*, pages 52–59. IEEE Computer Society Press. 10

[Frakes and Fox, 1996] Frakes, W. and Fox, C. (1996). Quality improvement using a software reuse failure modes model. *Software Engineering, IEEE Transactions on*, 22(4):274–279. 10

[Frakes and Gandel, 1990] Frakes, W. and Gandel, P. (1990). Representing reusable software. *Information and Software Technology*, 32(10):653–664. 11

[Frakes and Nejmeh, 1986] Frakes, W. and Nejmeh, B. (1986). Software reuse through information retrieval. In *ACM SIGIR Forum*, volume 21, pages 30–36. ACM. 11

[Frakes and Terry, 1994] Frakes, W. and Terry, C. (1994). Reuse level metrics. In *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pages 139–148. IEEE. 10

[Frakes and Terry, 1996] Frakes, W. and Terry, C. (1996). Software reuse: metrics and models. *ACM Computing Surveys (CSUR)*, 28(2):415–435. 9

[Fujimoto, 2000] Fujimoto, R. M. (2000). *Parallel and distributed simulation systems*. Wiley Series on Parallel and Distributed Computing. J Wiley & Sons. 19, 26

[Gaffney Jr and Durek, 1989] Gaffney Jr, J. and Durek, T. (1989). Software reuse–key to enhanced productivity: Some quantitative models. *Information and Software Technology*, 31(5):258–267. 10

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 36, 39

[Garlan and Perry, 1995] Garlan, D. and Perry, D. E. (1995). Introduction to the special issue on software architecture. *IEEE Trans. Softw. Eng.*, 21(4):269–274. 36

[Garlan and Shaw, 1994] Garlan, D. and Shaw, M. (1994). Characteristics of higher-level languages for software architecture. Technical Report CMUCS-94-210, School of Computer Science and Software Engineering Institute, Carnegie Mellon University. 13

[Genssler, 2002] Genssler, T. (2002). Pecos in a nutshell. www.pecos-project.org [Last accessed: 11/30/2010]. 13

[Gruber et al., 2005] Gruber, O., Hargrave, B. J., McAffer, J., Rapicault, P., and Watson, T. (2005). The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal*, 44(2):289–299. 34

[Gulyas and Kozsik, 1999] Gulyas, L. and Kozsik, T. (1999). The Use of Aspect-Oriented Programming in Scientific Simulations. In *Proceedings of Sixth Fenno-Ugric Symposium on Software Technology, Estonia*. 26

[Harzallah et al., 2008] Harzallah, Y., Michel, V., Liu, Q., and Wainer, G. (2008). Distributed simulation and web map mash-up for forest fire spread. In *IEEE Congress on Services-Part I, 2008.*, pages 176–183. IEEE. 23, 92

[Himmelspach, 2009] Himmelspach, J. (2009). Toward a collection of principles, techniques, and elements of simulation tools. In *Proceedings of the First International Conference on Advances in System Simulation*. IEEE Computer Society. 40

[Himmelspach et al., 2008] Himmelspach, J., Ewald, R., and Uhrmacher, A. M. (2008). A flexible and scalable experimentation layer. In Mason, S., Hill, R., Mönch, L., Rose, O., Jefferson, T., and Fowler, J., editors, *Proc, of the 2008 Winter Simulation Conference (WSC'08)*, pages 827–835, Miami, FL. 26, 86, 118

[Himmelspach and Uhrmacher, 2007] Himmelspach, J. and Uhrmacher, A. (2007). Plug'n simulate. In *40th Annual Simulation Symposium, 2007. ANSS'07*, pages 137–143. 22

[Himmelspach and Uhrmacher, 2009a] Himmelspach, J. and Uhrmacher, A. (2009a). The JAMES II Framework for Modeling and Simulation. In *2009 International Workshop on High Performance Computational Systems Biology*, pages 101–102. IEEE. 22

[Himmelspach and Uhrmacher, 2009b] Himmelspach, J. and Uhrmacher, A. M. (2009b). What contributes to the quality of simulation results? In Lee, L. H., Kuhl, M. E., Fowler, J. W., and Robinson, S., editors, *Proceedings of the 2009 INFORMS Simulation Society Research Workshop*, pages 125–129, University of Warwick, Coventry, U.K. INFORMS Simulation Society. 38, 40

[Holman et al., 2009] Holman, K., Kuzub, J., Moallemi, M., and Wainer, G. A. (2009). Cable-anchor robot implementation using embedded cd++. In Dalle, O., Wainer, G. A., Perrone, L. F., and Stea, G., editors, *SimuTools*, page 35. ICST. 92

[John, 1994] John, J. (1994). Marciniak, Encyclopedia of software engineering. 8

[Johnson, 2002] Johnson, D. (2002). A theoretician's guide to the experimental analysis of algorithms. In *Fifth and Sixth DIMACS Implentation Challenges*. 2

[Johnson and Foote, 1988] Johnson, R. E. and Foote, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35. 43

[Jones, 1993] Jones, C. (1993). Software return on investment preliminary analysis. *Software Productivity Research, Inc.* 8

[Karlsson, 1995] Karlsson, E. (1995). *Software reuse: a holistic approach*. John Wiley & Sons. 8

[Karunanithi and Bieman, 1993] Karunanithi, S. and Bieman, J. (1993). Candidate reuse metrics for object oriented and Ada software. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 120–128. IEEE. 10

[Kelton and Law, 2000] Kelton, W. and Law, A. (2000). *Simulation modeling and analysis*. McGraw Hill. 2

[Kernighan, 1984] Kernighan, B. (1984). The Unix system and software reusability. *Software Engineering, IEEE Transactions on*, (5):513–518. 11

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An overview of AspectJ. *European Conference on Object-Oriented Programming, 2001,ÄîObject-Oriented Programming*, pages 327–354. 33

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European Conference on Object-Oriented Programming, ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland. Springer-Verlag. 32, 74

[Koltun and Hudson, 1991] Koltun, P. and Hudson, A. (1991). A reuse maturity model. 10

[Krueger, 1992] Krueger, C. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183. 8

[Kurkowski et al., 2005] Kurkowski, S., Camp, T., and Colagrosso, M. (2005). MANET simulation studies: the incredibles. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(4):50–61. 2

[Lacage, 2010] Lacage, M. (2010). *Outils d'expérimentation pour la recherche en réseaux*. PhD thesis, Université de Nice. 25, 92

[Lau and Wang, 2005] Lau, K.-K. and Wang, Z. (2005). A taxonomy of software component models. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, EUROMICRO '05, pages 88–95, Washington, DC, USA. IEEE Computer Society. 14

[Law, 2007] Law, A. M. (2007). *Simulation Modeling and Analysis*. McGraw-Hill International, 4 edition. 16, 17, 18, 37, 40, 72, 73

[Leclercq et al., 2005] Leclercq, M., Quéma, V., and Stefani, J.-B. (2005). DREAM: a Component Framework for the Construction of Resource-Aware, Configurable MOMs. *IEEE Distributed Systems Online*, 6(9). 77

[L'Ecuyer, 1990] L'Ecuyer, P. (1990). Random numbers for simulation. *Commun. ACM*, 33(10):85–97. 2

[L'Ecuyer et al., 2002] L'Ecuyer, P., Meliani, L., and Vaucher, J. (2002). SSJ: SSJ: a framework for stochastic simulation in Java. In *Proceedings of the 34th conference on Winter Simulation Conference: exploring new frontiers*, page 242. Winter Simulation Conference. 3

[Lim, 1994] Lim, W. (1994). Effects of reuse on quality, productivity, and economics. *Software, IEEE*, 11(5):23–30. 8

[Luby et al., 1997] Luby, M., Mitzenmacher, M., Shokrollahi, M., Spielman, D., and Stemann, V. (1997). Practical loss-resilient codes. In *Proc. ACM Symp. on Theory of computing*, pages 150–159. 96

[Madsen, 2003] Madsen, K. (2003). Five years of framework building: lessons learned. In *18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 345–352. ACM. 43

[Margono and Rhoads, 1992] Margono, J. and Rhoads, T. (1992). Software reuse economics: cost-benefit analysis on a large-scale Ada project. In *Proceedings of the 14th international conference on Software engineering*, pages 338–348. ACM. 10

[McIlroy et al., 1969] McIlroy, M., Buxton, J., Naur, P., and Randell, B. (1969). Mass produced software components. *Software Engineering Concepts and Techniques*, pages 88–98. 8, 13

[Medvidovic and Taylor, 2002] Medvidovic, N. and Taylor, R. (2002). A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93. 13, 14

[Mili et al., 1995] Mili, H., Mili, F., and Mili, A. (1995). Reusing software: Issues and research directions. *Software Engineering, IEEE Transactions on*, 21(6):528–562. 8

[Moallemi and Wainer, 2009] Moallemi, M. and Wainer, G. A. (2009). A system-on-chip fpga implementation of embedded cd++. In Wainer, G. A., Shaffer, C. A., McGraw, R. M., and Chinni, M. J., editors, *SpringSim*. SCS/ACM. 92

[Mohagheghi and Conradi, 2008] Mohagheghi, P. and Conradi, R. (2008). An empirical investigation of software reuse benefits in a large telecom product. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(3):1–31. 10

[Moore, 1994] Moore, J. (1994). Debate on software reuse libraries. In *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pages 203–204. IEEE. 12

[Morisio et al., 2002] Morisio, M., Ezran, M., and Tully, C. (2002). Success and failure factors in software reuse. *IEEE Transactions on software engineering*, pages 340–357. 10

[Pawlikowski et al., 2002] Pawlikowski, K., Jeong, H.-D., and Lee, J.-S. (2002). On credibility of simulation studies of telecommunication networks. *Communications Magazine, IEEE*, 40(1):132–139. 2, 39

[Pawlikowski and Yau, 1993] Pawlikowski, K. and Yau, V. (1993). AKAROA: a package for automatic generation and process control of parallel stochastic simulation. *Australian Computer Science Communications*, 15(1):71–82. 3

[Pearce, 2003] Pearce, T. (2003). Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems. In *Real-Time Systems Symposium. Work-in-Progress Session. Cancun, Mexico*. 23

[Perrone et al., 2009] Perrone, L. F., Cicconetti, C., Stea, G., and Ward, B. (2009). On the automation of computer network simulators. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (SIMUTools 2009)*, pages 49:1–49:10. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), Brussels, Belgium. 118

[Poulin et al., 1993] Poulin, J., Caruso, J., and Hancock, D. (1993). The business case for software reuse. *IBM Systems Journal*, 32(4):567–594. 10

[Quema, 2005] Quema, V. (2005). *Vers l'exogiciel–Une approche de la construction d'infrastructures logicielles radicalement configurables*. PhD thesis, Institut National Polytechnique de Grenoble. 61

[Ribault et al., 2010] Ribault, J., Dalle, O., Conan, D., and Leriche, S. (2010). OSIF: a framework to instrument, validate, and analyze simulations. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, pages 1–9. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 72

[Robinson, 2004] Robinson, S. (2004). *Simulation: The practice of model development and use*. John Wiley & Sons. 17

[Röhl and Uhrmacher, 2008] Röhl, M. and Uhrmacher, A. M. (2008). Definition and analysis of composition structure for discrete-event models. In Mason, S., Hill, R., Moench, L., and Rose, O., editors, *Proceedings of the Winter Simulation Conference*, pages 942–950. 11

[Romero et al., 2009] Romero, D., Rouvoy, R., Chabridon, S., Conan, D., Pessemier, N., and Seinturier, N. (2009). *Enabling Context-Aware Web Services: A Middleware Approach for Ubiquitous Environments*. Chapman and Hall/CRC. 77, 85

[Rouvoy et al., 2008] Rouvoy, R., Conan, D., and Seinturier, L. (2008). Software Architecture Patterns for a Context Processing Middleware Framework. *IEEE Distributed Systems Online*, 9(6). 73, 74, 77

[Rouvoy et al., 2006] Rouvoy, R., Pessemier, N., Pawlak, R., and Merle, P. (2006). Using attribute-oriented programming to leverage fractal-based developments. In *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal,Äô06), Nantes, France*. 90

[Sametinger, 1997] Sametinger, J. (1997). *Software engineering with reusable components*. Springer Verlag. 8, 11

[Sargent, 2008] Sargent, R. G. (2008). Verification and validation of simulation models. In Mason, S. J., Hill, R. R., Mönch, L., Rose, O., Jefferson, T., and Fowler, J. W., editors, *Proceedings of the 2008 Winter Simulation Conference*, pages 157–169, Piscataway, New Jersey. Institute of Electrical and Electronics Engineers, Inc. 2, 37

[Schäfer et al., 1993] Schäfer, W., Prieto-Díaz, R., and Matsumoto, M. (1993). *Software reusability*. Ellis Horwood. 8

[Seinturier et al., 2005] Seinturier, L., Pessemier, N., and Coupaye, T. (2005). AOKell: An aspect-oriented implementation of the Fractal specifications. Objectweb Fractal Workshop, Grenoble, France. 24

[Seinturier et al., 2006] Seinturier, L., Pessemier, N., Duchien, L., and Coupaye, T. (2006). A component model engineered with components and aspects. *Component-Based Software Engineering*, pages 139–153. 29

[Selby, 1989] Selby, R. (1989). Quantitative studies of software reuse. In *Software reusability*, pages 213–233. ACM. 11

[Seo, 2006] Seo, H. S. (2006). Network security agent DEVS simulation modeling. *Simulation Modelling Practice and Theory*, 14(5). doi:10.1016/j.simpat.2005.08.010. 55

[Sommerville, 2007] Sommerville, I. (2007). *Software Engineering*. Addison-Wesley, 8 edition. 2, 8, 36, 40, 43

[Spinczyk et al., 2002] Spinczyk, O., Gal, A., and Schröder-Preikschat, W. (2002). AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, page 60. Australian Computer Society, Inc. 33

[Stafford and Urbaczewski,] Stafford, T. and Urbaczewski, A. Spyware: The ghost in the machine. *Communications of the Association for Information Systems (Volume14, 2004)*, 291(306):291. 53

[Szyperski, 2002] Szyperski, C. (2002). *Component Software - Beyond Object-Oriented Programming*. AW, 2nd edition. 13, 29

[Taivalsaari and Jyväskylän, 1993] Taivalsaari, A. and Jyväskylän, Y. (1993). *A critical view of inheritance and reusability in object-oriented programming*. University of Jyväskylä. 8

[Tracz, 1988] Tracz, W. (1988). Software reuses: motivators and inhibitors. In *Software reuse: emerging technology*, pages 62–67. IEEE Computer Society Press. 8

[Troitzsch, 2004] Troitzsch, K. G. (2004). Validating simulation models. In Horton, G., editor, *18th European Simulation Multiconference. Networked Simulations and Simulation Networks*, pages 265–270. The Society for Modeling and Simulation International, SCS Publishing House. 2

[Van Ommering et al., 2002] Van Ommering, R., van der Linden, F., Kramer, J., and Magee, J. (2002). The Koala component model for consumer electronics software. *Computer*, 33(3):78–85. 15

[Van Waveren et al., 2000] Van Waveren, R. H., Groot, S., Scholten, H., Van Geer, F., Wösten, H., Koeze, R., and Noort, J. (2000). *Good Modelling Practice Handbook.* STOWA, Utrecht, RWS-RIZA, Lelystad, The Netherlands. 37

[Varga, 2001] Varga, A. (2001). The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic. 24, 48

[Varga and Hornig, 2008] Varga, A. and Hornig, R. (2008). 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). 26, 48, 73, 83

[Wainer, 2002] Wainer, G. (2002). CD++: a toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306. 23

[Wang and Lehmann, 2008] Wang, Z. and Lehmann, A. (2008). Expanding the V-Modell XT for verification and validation of modelling and simulation applications. In *System Simulation and Scientific Computing*, pages 404–410. Asia Simulation Conference. 37

[Washizaki et al., 2003] Washizaki, H., Yamamoto, H., and Fukazawa, Y. (2003). A metrics suite for measuring reusability of software components. 10

[Weatherspoon and Kubiatowicz, 2002] Weatherspoon, H. and Kubiatowicz, J. (2002). Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, pages 328–338. 96

[Webster, 1995] Webster, B. F. (1995). *Pitfalls of object-oriented development.* M&T Books, New York. 3

[Zeigler, 1976] Zeigler, B. P. (1976). *Theory of Modelling and Simulation.* Wiley & Sons, NY. 22, 49

[Zeigler, 1984] Zeigler, B. P. (1984). *Multifacetted Modelling and Discrete Event Simulation.* Academic Press Inc., London. 22, 26, 72, 90

[Zeigler et al., 2000] Zeigler, B. P., Kim, T. G., and Praehofer, H. (2000). *Theory of Modeling and Simulation.* Academic Press, Inc. 4, 17, 23, 49

Abstract

Studying a system using discrete-event computer simulations implies several activities: conceptual model specification, software model architecture description, software development, simulation scenario, instrumentation, experimentation planning, computational resources configuration, execution, post-processing and analysis, validation and verification (V&V). Many software are required to complete all these activities. However, it is common practice to create a simulator from scratch when starting a new a simulation study. It is therefore necessary to redevelop a whole suite of tools to ensure support for all simulation activities.

This thesis addresses the challenge of developing new simulators that reuse existing models and simulator parts. Indeed, reusing software increases dependability, is less error prone, makes better use of complementary expertises, improves standards compliance, and accelerates development. Reusing software can be applied to all simulation activities. Several problems have to be solved to derive full benefit of reuse. In this thesis, we address three major issues: Firstly, we investigate practical means of reusing and combining valuable pieces of modeling and simulation software at large, including models, simulation engines and algorithms, and supporting tools for the modeling and simulation methodology; Secondly, we focus on issues related to instrumentation; Thirdly, we focus on problems of integration of existing simulation tools.

To achieve these objectives, we investigate advanced software engineering techniques such as component-based software engineering (CBSE) and aspect-oriented programming (AOP), and use them to derive a novel approach for Modeling & Simulation based on reusable layers. We developed a prototype software architecture that proves the feasibility of this layered approach.

Keywords: simulation, discrete events, aspects, separation of concerns, instrumentation, modeling, component, distributed simulation, reuse