HAL
open science

Should Simulation Products use Software Engineering Techniques or Should they Reuse Products of Software Engineering? – Part 1

Olivier Dalle

To cite this version:

Olivier Dalle. Should Simulation Products use Software Engineering Techniques or Should they Reuse Products of Software Engineering? – Part 1. SCS Modeling and Simulation Magazine, 2011, 2 (3), pp.122-132. inria-00638553

HAL Id: inria-00638553

https://inria.hal.science/inria-00638553v1

Submitted on 5 Nov 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Should Simulation Products use Software Engineering Techniques or Should they Reuse Products of Software Engineering? – Part 1

Olivier Dalle

Université Nice Sophia Antipolis
Laboratoire I3S UMR CNRS 6070
INRIA Sophia Antipolis - Méditerranée
2004, Route des Lucioles,
Sophia Antipolis, France.

ABSTRACT

This two-part article addresses the issues concerning the building of new simulation software by either reusing existing general purpose software products and frameworks or by writting the simulation software from scratch. As a means of discussing the use of existing software, this first part describes a selected list of such existing software: the Eclipse IDE as graphical user front-end, Maven for the management and building of projects, Bonita for supporting simulation workflows, Ruby on Rails and its Hobo extension to provide online persistence, and the Fractal Component Model for supporting the popular Component-Based Modeling & Simulation approach. The second part, to be published in the next issue of the *M&S Magazine*, will further explore some interesting features found in the selected software solutions, and discuss their benefits when applied to simulation.

1 Introduction

The presentation that follows is the first of a two-part article. The Part One will outline a number of existing software products and will show that they are good candidate for integration in a new simulation software. The sequel article (to appear in the October issue of the *M&S Magazine*) will report on a practical experience in using these products to build a new simulation software product, and will discuss some lessons learned from this approach.

Building a new simulation software is a very popular exercise. Indeed, when considering all the applications of simulation, such as gaming, scientific studies, military applications and many more, a significant number of computer scientists and software engineers have certainly been involved in the development of a simulator at some point in their programming experience. Unfortunately, what might appear as an easy project at the early stage of the devlopment often ends up being an endless source difficulties and eventually a time-consumming (and cost-consumming)

exercise. However, underestimation is certainly not the only reason that explains why starting such a development is so popular. Albeit reinventing the wheel is often considered a waste of time, it actually turns to have some benefits[5]. For example, reinventing the wheel is an excellent way of learning how things work, and why some solutions are better than other.

Out of the many reasons that might motivate this decision, a common one is that, at some point, existing simulators are not good enough, either because the problem that motivate the simulation is new or different, or because new techniques have emerged from software engineering and M&S that made the *old*(-fashioned) simulators obsolete.

In this paper, discuss ideas for efficiently building yet another good simulation software. More precisely, the scope of my discussion is limited to intertwined aspects of Software Engineering and Simulation. Indeed, building a good simulation product is also a question of design, ergonomics, and is dependent on the functionnal coverage of the software: The less a software has functionnalities, the less it takes the risk of disppointing its end-users (although there is an obvious lower limit to this principle).

Leaving apart the question of which functionnality to provide, the actual question addressed in this paper is the following: Given a set of (agreed) functionnalities, how can we design a good simulation product to provide these functionalities? A very popular approach, if not the most popular, consists in building a new product from scratch, while using the most up to date Software Engineering techniques of the moment. At the time of writing, good candidates techniques include Model Driven Approach, Test Driven Development, Component-Based Software Engineering and the likes. Undoubtedly, those are good and exciting techniques, but once the technique have been carefully chosen, the time and effort it takes to market (or release) the resulting product is still in the order of several years.

The different approach discussed in this article consists in building a new product by reusing already existing products and implemented techniques. This is actually no less than implementing a form of reuse, which again, is not so original. Indeed, reuse has long been popular in the simulation community (see for example DEVS, proposed by Zeigler in 1976), and many efforts have been made to build products that allow for reuse. Component-Based Modeling & Simulation (CBMS), such as with DEVS, is generally accepted as a good way to achieve reuse, but other techniques exist, such as Agent-Based Simulation, middleware RTIs such as HLA or, more recently, the Web-Based mash-ups[**?**]. In the case of CBMS, the underlying idea is that many products implement CBMS so that the component developed within such a product can be reused by others in the same product, or even in other products. However, even with a non-ambiguous formalism, such as the one used in DEVS, reuse of DEVS components is still an open issue in the community, despite the formalism has existed since the 70's. Going one step further, SISO did put a significant effort in standardizing simulation model components with the BOMS standard. BOMS is a significant contribution that includes a number of interesting ideas, but it still does not solve fully the issues of reuse, in particular on semantics aspects.

The reuse approach promoted in this article is slightly different. Instead of just enabling reuse of models or simulation domain-specific code, why not just reuse general purpose products or techniques that are already implemented,

widely available, used, and well documented? This might require some adaptation, but we can still expect the required development effort to be smaller than redeveloping an equivalent product exclusively geared at simulation. Reusing a mature, general purpose product has plenty benefits. First, since it is mature, it certainly went through several iterations of bug fixes and improvements and already has a community of experienced users. Second, because it was designed to be general purpose, it may come with its own self-contained philosophy, which might be different from what would first come in mind when implementing a simulation-specific product, but prove to proficiently serve the product.

Assuming this approach is worth to try, the next question is: What good products and implemented techniques are available for reuse and how can we reuse these products to build a (good) simulation product? The answer to the latter question is certainly not unique and depends a lot on the current trends.

In the following Part One of this article, a few examples of some *trendy* implemented techniques or products will be given. Going one step further, in the Part Two of this paper, the way they could be proficiently reused for building a good product for simulation will be illustrated. In Part Two, new ideas derived from these trends will be discussed, and the way they can be adapted or ported with benefits to future simulation product developments will be demonstrated.

The wide variety of software products that fall under the appelation of Simulation Software makes it difficult to accurately define the minimal set of functionnalities required in simulation software. However, a reasonable set of features that might be found in a modern simulation products could include the following:

- Front-end User Interface: A graphical user interface to support user interactions;
- Project Management: A system for building complex projects with dependencies;
- Management of Workflows: A machinery to support simulation methodology through workflows;
- Online Database: A solution for implementing the persistence of models, experiments, and results and make them available to the scientific community;
- Component-Based Modeling: A framework for the composition of models.

This list could easily be augmented with other important features, such as visualization, versioning, or validation & testing, but the purpose of this paper is not to be exhaustive. For each of the afore mentioned features, one ready-to-use solution is suggested. Here again, more solutions might exist, but the point of this discussion is to show that at least one such solution exists and provides a potentitally good support building simulation software. In Section 2, the potential use of Eclipse for providing the Front-End User Interface is discussed; In Section 3, the use of Maven for dealing with project management issues and building complex projects is proposed; In Section 4, the potential use of Bonita for supporting the simulation workflows is explained; In Section 5, Ruby-on-Rails and its Hobo extension as a means for implementing an online database and provide a RESTful persistence service is described; Lastly, in Section 6, the Fractal Component Model as a means for providing Component-Based Modeling described.

2 Front-end User Interface

Using and extending an already existing Integrated Development Environment (IDE) is certainly not a novel idea. On the contrary, many simulators already reuse or extend existing general purpose IDEs. The **Eclipse** IDE (available from www.eclipse.org) is particularly popular for this purpose for two reasons: First it is highly custumizable because of its all-as-a-plug-in architecture, and second it leaves the choice to the end-users of configuring their own environment according to their needs, by selecting the set of plug-ins they want to use among the large collection of available Eclipse plugins, and possibly more importantly, by removing the ones they don't like. Indeed, in terms of productivity, being able to skip or replace badly designed features can already result in significant improvements.

The ability of the Eclipse platform to deliver domain-specific functionnalities can even be further extended to the point where the IDE itself is entirely dedicated for a particular application. In this case, the platform becomes a so-called "Rich-Client," whose aim is no longer to let people add or remove plugins, but to provide a fully integrated solution, prepared and optimized for a particular application, usually by a vendor. The Omnet++ simulator is an example of such an extreme customization, as shown on figure 1.

Hence, from a simulation product designer's point of view, Eclipse offers an interesting range of possibilities: At one extreme, one can start modestly by contributing a single plug-in covering a particular aspect of the domain-specific application, while at the other extreme, one can create a fully dedicated, self-contained application. It is also worth mentioning that Eclipse plug-ins offer extension points, which means

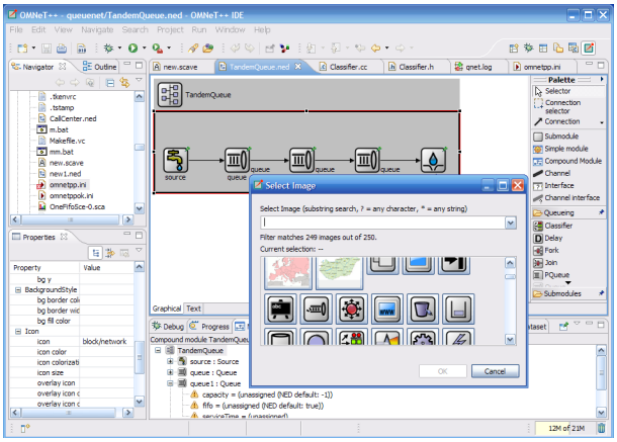

Figure 1: A screenshot of the Omnet++ Eclipse-based Rich Client User Interface (more available on www.omnetpp.org).

that existing general purpose plug-ins can be extended or specialized for more specific purposes. For example, a general purpose XML editor could be extended to support an XML-based Domain Specific Language.

Eclipse also comes with a rich-featured Plugin Development Environment (Eclipse PDE) and a number of Frameworks, such as the EMF, GEF or GMF (respectively the Eclipse Modeling, Graphical Editing, and Graphical Modeling Frameworks) that allow the rapid development of eclipse extensions and plug-ins.

Coming back to our discussion, what are the benefits of reusing an IDE such as Eclipse compared to building a dedicated user interface from scratch? First of all, considering its large number of existing plugins, Eclipse provides an extensive support for almost all the existing programming languages. Furthermore, this support can be extended, adjusted, or replaced with a number of alternatives, and new specific support can be built from existing plug-ins, which speeds-up the

development. Eclipse also supports well graphical modeling and notations and provides a way of building specialized views for particular tasks. Without an IDE such as Eclipse, nonetheless all these features would have to be redeveloped, but they would also have to suffer the comparison: Eclipse is well known, stable, and almost established as standard. Hene, providing a lower than standard support would certainly not contribute to the popularity of a new product.

3 Project Management

Maven is a project supported by the Apache Foundation. The objectives of maven are the following (quoting maven.apache.org):

- "Making the build process easy."
- "Providing a uniform build system."
- "Providing quality project information."
- "Providing guidelines for best practices development."
- "Allowing transparent migration to new features."

What makes Maven interesting is the fact that it is the result of developers' experience. Indeed, the Apache Foundation hosts many developments, some of which require a complex building machinery and project management. Hence, realizing the limitations of *Ant*, the traditional building tool for Java-based projects, a new tool was built internally to serve some of the Foundation's projects. Realizing the value of their tools, the developers of Maven decided to release it officially as an Apache Foundation project. At the time that Maven was first released, it had already gone through a number of improvement iterations resulting from its actual use in major development projects.

Listing 1: A sample Maven POM file (copied from the tutorial on maven.apache.org).

```
<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/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
  <properties>
    <my.filter.value>hello</my.filter.value>
  </properties>
</project>
```

Delving a bit more into the details, the starting point in Maven is an XML-based project description file called a POM (Project Object Model). Such a POM file (named `pom.xml`) is associated to each sub-component of the project.

This file contains all the necessary information to deal with the component life-cycle: How to generate its documentation, what helpers are needed to build its code, run tests, produce reports (eg. test coverage), describe its dependencies to other sub-components, define the current version number, deploy the code, deploy the documentation on a dedicated website, save updates on a Source Code Management system (eg. SVN or CVS), interface with a bug-tracking system, and so on. While this latter enumeration

is quite disorganized, Maven is all the contrary: It provides a strict file layout for each project (new types of projects being defined by means of "archetypes") and everything in Maven is designed so as to avoid unexpected situations that could cause a failure to build the project. For example, Maven does not allow a (sub-)project to be locally customized such that it becomes permanently linked to a particular execution context (eg. to a specific host machine or user account). This comes at a price in terms of flexibility, because users are not given the option to customize an *existing* project as much as they might want. However, it is worth noting that (i) in real life the same price has to be paid when working collectively under the supervision of a (human) project manager (ie. a project manager is expected to enforce common rules in the development team), and (ii) Maven fully supports dependencies, which makes it easy to derive a new project from an existing one.

Maven also includes a bit of magic: it downloads and compiles all the required dependencies, including most of itself from online repositories. In practice, the only things required to build a Maven project from scratch are the Maven command (`mvn`), or its integration plugin for Eclipse, and the source files (including the pom file). Everything else, including compilers and tools, is downloded and possibly recompiled by Maven on the fly. This makes the first compilation a bit lengthy, because it may trigger the downloading and recompilation of tens of packages, but it leads to another astonishing observation: Maven downloads and recompilation are pretty fast. Furthermore, a local cache significantly improves the performances of subsequent recompilations of a project.

Compared to a proprietary integrated solution developed from scratch, or even compared to a solution based on classic development tools such as `Ant` or `make`, the benefits of using Maven is to cover the project management at large. Indeed, `make`, `ant` and the likes are only providing support for building a project. Most of them do not even provide the automatic analysis and download of dependencies, although the *configure* scripts help at least to find out what is missing. Nonetheless Maven finds what is missing but it finds and installs it proprerly and quickly. And it provides many more, as mentioned earlier. In the particular context of scientific simulations, the fact that Maven is a network-centric tool is also invaluable, because it provides an easy an well tested means for putting the simulations online, and help to reproduce scientific results.

4 Management of Workflows

Bonita is the first of two products described in this paper that emerged from the ObjectWeb Consortium developments (OW). The OW developments are open source projects geared at providing middleware solutions for grid, cloud, and general purpose distributed computing (see www.ow2.org). Bonita is a set of three tools available as Open Source projects, but also released by a vendor under the name Bonita Open Solution (BOS). The three tools included in BOS are: A studio to design workflows using the Business Process Model And Notation formalism (BPMN), an Engine to run the BPMN workflows, and a web-based user interface for users to manage and interact with workflows (albeit, this latter interface might not be necessary in a simulation environment if the workflow is supposed to be hidden). With BPEL, BPMN is one of the standardized formalisms for the specification of

Business Processes; version 2.0 of the standard was approved by the OMG (Object Management Group) in January 2011.

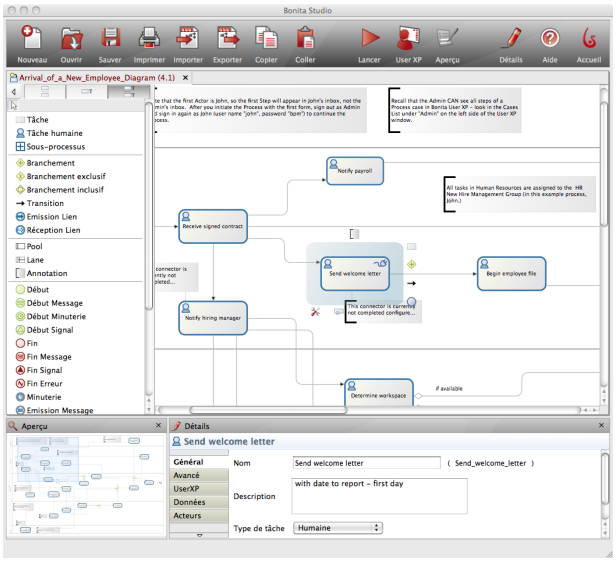

Figure 2: A screenshot of the Bonita Studio interface.

Let us have a closer look at BPMN. As shown on Figure 2, BPMN is a flow-chart notation that offers four kinds of elements: Flow objects (activities, events, gateways), connectors (sequence flow, message flow and associations), artifacts (data objects, text annotations, groups) and swimlanes (a workflow is contained in a *pool* that can be divided in multiple *lanes* representing flows progressing in parallel, like swimmers in a pool). Without entering the details of the notation and Bonita's implementation (see www.bonitasoft.org for more information), an interesting feature of Bonita lies is found in the connectors it offers. Indeed, in a typical simulation workflow, in addition to the usual flowchart expressions, an important requirement is the ability to keep the users in the loop and let them interact easily with the workflow when required, or let the workflow proceed automatically when some end-user feedback is not necessary. For this purpose, Bonita has taken care in providing an extensible and well documented connector definition API. Hence, new connectors can be added to match the particular requirements of the application. For example, such connectors could be linked to electronic mail, social messaging, file storage, web services, enterprise content delivery systems, and so on.

Compared to a solution developed from scratch, Bonita already provides all the necessary support for expressing and managing workflows. Nonetheless using an existing product like Bonita saves a significant amount of development time, but it also prevents to suffer from a number of failure that would undoubtedly result from the use of a new and unsufficiently tested development. Indeed, coming back to simulations, it should be kept in mind that a single error in a workflow engine might invalidate thousands of hours of computation. Despite Bonita might still suffer from a few bugs, its already long history gives it a clear advantage compared to a newly developed solution, in addition to the development time saved by reusing the product.

5 On-line Database

Ruby on Rails (RoR) is a very popular framework for developing RESTful databases. As its name suggests, it is based on the Ruby language, which is not so common, but has actually some advantages. Indeed, Ruby (as well as Python and a few others), are so-called "prototyping-language" because they were specially designed and optimized for the rapid prototyping of applications. Indeed, with such a languages, a pro-

totype of a new application, including its Graphical User Interface can be developped in very short time. Ruby is very well designed, and supports advanced features such as object-oriented programming, mixins, functionnal programming and many more. The Rails framework, that gave its name to RoR, adds to Ruby advanced DataBase support.

Rails implements the well-known Model-View-Controller design pattern[3] (MVC), which connects a Database to a user interface through a set of controllers: The user interface is made of Views, which only contain the code for presenting the data to users; the controller contains the logic of the interactions of the users with the Database, and the Model describes the Database schema and its relational and transactional logic.

To the MVC pattern, which is not a new idea, Rails adds a collection of design ideas and principles that have recently emerged from the experience of developers in software industry. The first of these design principles is "DRY", which means "Don't Repeat Yourself". Indeed, in Rails everything is done so that you don't have to repeat again and again the same coding idioms. This is partly achieved thanks to a set of generator scripts, that automate the generation of recurring code when it cannot be factorized.

Once the code is generated, then we enter into the scope of another interesting principle of Rails: Promote conventions over configurations. Rails provides reasonable defaults for everything: When the code of the application is generated, it is ready for production without any configuration. Then, the default behavior can be changed wherever it is deemed necessary.

The third principle is to rely on a RESTful web interface. REST is a design pattern for web applications that fully exploits the potential semantic of URIs and HTTP verbs[2]. In brief, HTTP provides four verbs (GET, PUT, POST, DELETE) each of which associated with a particular action that applies to a *resource* pointed by a URI, according to a routing mechanism. For example, the "GET" action applied to the URI "server.org/customer/2" would retrieve the second record of the table that stores the resource "customer" in the database, while the "POST" action on the same URI could be used to update that same record. Therefore, interacting with a database becomes almost as easy as typing a URI in a web browser (at least for GET actions).

The most interesting feature of RoR is certainly its ability to support incremental development. RoR provides a "migration" mechanism that contains the necessary "up" code for migrating the database schema from one version to the next one. This mechanism also contains the "down" code to undo the last modification and rollback to the last version of the database. This incremental approach dramatically changes the development process. Indeed, instead of spending months in order to achieve a perfect database design, RoR lets the developer play with database schema and make changes to the application on the fly, as new bugs or missing parts are discovered.

RoR also includes many invaluable features. For example, it comes with three databases: one is for development, another is for the code already in production and a testing database. Altogether these three database allow to put the application in production early in the development process, while developments are still ongoing, because new untested features are safely kept separated from the production version, in the developement database.

RoR can be used as is, but it is still a bit tedious. **Hobo** is a framework built on top of Rails that adds even more DRY concepts. For

example, in Rails, the Views of the MVC pattern are implemented using "erb" files, which are HTML page snippets with embedded-Ruby code, ie. ruby code that is dynamically interpreted. In Hobo, the erb files are replaced with "DRYML" files, an xml-based language that is specially optimized to enforce the DRY and convention over configuration principles. In practice, the results are astonishing, and a few lines of DRYML code are usually enough to make all the necessary changes required by an application.

Compared to a proprietary Database solution developed from scratch, RoR and Hobo provide a means for the rapid creation of a RESTful database. Furthermore this creation process can be made incremental, while a traditionnal Database development usually require to explicit the full database schema at the beginning of the development, in order to explicit the relations that need to backed-up in the supporting code. Last but not least, a traditional development of a Database that has a web interface may require the tedious writing of numerous HTML forms. With Hobo, this step is much easier, if not anecdotic.

6 Component-Based Modeling

The **Fractal Component Model** (FCM)[1] is the second of the two products supported by the ObjectWeb consortium presented in this article. FCM provides support for building component-based applivations. It also provides means for applying the Separation of Concerns (SoC) Software Engineering principle.

First, it provides an Architecture Description Language (ADL) and advanced mechanisms for building component architectures, such as *factories* or *template components*. Thanks to the ADL, the concern of building the topological description of the hierarchy of components is separated from other concerns. Factories are special components that can dynamically instantiate new components. Therefore, the way components are instantiated may be implemented in a self contained component, which is a means of separating the instantiation concern from others. The default FCM ADL parser *is* a hierarchical factory component. Template components are special factory components that may be used to build a generic model of hierarchical components. Such template models may then be used to instantiate homomorphic copies of the model.

Second, FCM offers good support for non-functional concerns. This framework consists in embedding each component into a software *membrane*: the content part of the component implements its functional concerns, and the membrane part implements its non-functional concerns. The membrane consists of several *controllers*, each of which is responsible for a non-functional concern (figure 3). The framework allows for the construction of new membranes by assembling new or existing controllers. The selection of which membrane to associate with which content may be specified using the ADL.

Interestingly, FCM provides *shared components*. In a hierarchical component model, a shared component is a component that has more than one parent in the component hierarchy. To the author's knowledge, very few component models do effectively support the shared component feature: The Fractal component model does explicitly support sharing while some others, like JainSLEE[4] provide proxying techniques, which is a practical way of implementing sharing.

Unlike most other component models, FCM is not linked to a particular programming lan-

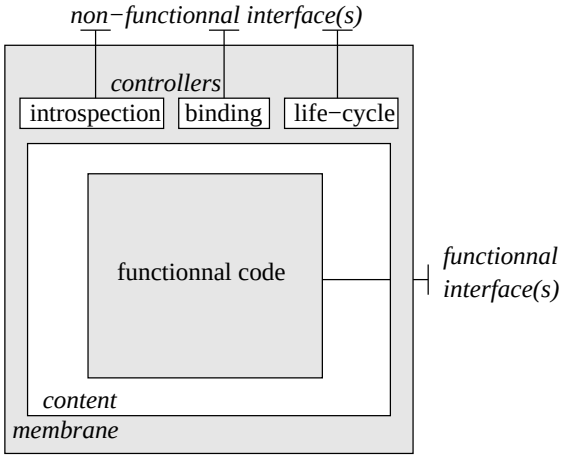

Figure 3: Anatomy of a Fractal component. In this example, the membrane contains three controllers that offers the introspection, binding and life-cycle of non-functional services.

guage. Indeed, FCM is a specification that may be implemented in multiple languages. Several implementations are already available or are under development, in different languages, such as Java, C++, C or SmallTalk, for example. No actual middleware implementation exists for the particular purpose of coupling Fractal components developed in these various languages, however non-FCM specific solution could be envisioned, such as through the use of web-services connectors between components.

Several solutions are available for the distributed execution FCM applications. For example, the FCM site provides a Java library called FractalRMI that transparently implements proxy/stub coupling between distributed components. Another library, called Fractal-BF (Binding Factory), allows the definition of advanced coupling between components, including distributed coupling. Since FCM is a specification, some integrated solutions such as the ProActive platform (distributed by ActiveEon) provides support for Distributed High Performance Computing on the Grid; ProActive implements the FCM specifications while offering advanced distributed services such as FireWall bypassing strategies or dynamic load balancing of workloads between computing nodes.

Building a proprietary component specification for a particular purpose requires a careful design, because once the design starts to be used, it becomes very difficult to change the specifications. Furthermore, using a well-known general purpose component model allows to reuse the wide range of tools and libraries contributed for this model, as is the case for FCM. Compared to other component models, FCM is one of the rare existing specification that supports hierarchical components, hence my choice. Indeed, supporting hierarchical components adds a significant complexity to the model, because it most provide means for building, updating and navigating through the hierarchy. Moreover, since the component model is the base for everything, it has to be one of the first developments: as long as the component specification is not released, no model can be developed. On the contrary, since the component are designed to separate the business logic of the models from the technical logic of simulation, the development of models can start almost as soon as the component specification is ready.

7 Concluding Remarks (Part1)

In this Part One, existing software solutions that seem to nicely cover some of the functional requirements of a typical simulation software have been described. Despite these solutions are

ready to use, glueing them together still requires some work, as will be discussed in more details in the Part Two of the paper. However, even when considering the issue of integrating multiple existing software or frameworks into a coherent application, and even when considering the increased effort of delving into other's works, documentations, and procedures, the overall benefits of this approach is worth the pain.

The time and effort required to achieve such a level of achievement when starting a new development from scratch is tremendous. In particular, the existing solutions that have been chosen in this paper have gone through an already long development history, made of numerous bugfixes and improvement releases, and based on the feed-back of very active Open Source communities.

Furthermore, as will be shown in the Part Two of this article, this long development history lead to a number of interesting features that happen to be very useful when applied to simulation, and may not have been considered if the development had started with only simulation requirements in mind.

ACKNOWLEDGMENTS

This work is partly founded by the French Agence Nationale de la Recherche (ANR), in the USS-SimGrid project and partly by INRIA, in collaboration with University of Carleton, in the context of the Associated Team DISSIMINET. The author addresses his special thanks to G. Wainer (Carleton University, Ottawa) and E. Mancini (INRIA, Sophia Antipolis) for their valuable comments and suggested corrections.

References

[1] E. Bruneton, T. Coupaye, M. Leclercq, V. Quma, and J.-B. Stefani. The fractal component model and its support in java, 2006.

[2] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[3] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] Swee Boon Lim and David Ferry. *Jain SLEE 1.0 Specification*. Sun Microsystems Inc. & Open Cloud Ltd., 2002.

[5] Ilan Oshri, Sue Newell, and Shan L. Pan. Implementing component reuse strategy in complex products environments. *Commun. ACM*, 50:63–67, December 2007.

AUTHOR BIOGRAPHIES

Olivier Dalle is *Maître de Conférences* in the Computer Science department of the Faculty of Sciences at the University of Nice-Sophia Antipolis (UNS). He received his B.Sc. from the University of Bordeaux 1 and his M.Sc. and Ph.D. from UNS. From 1999 to 2000, he was a postdoctoral fellow at the French Space Agency center in Toulouse (CNES-CST), where he started working on component-based discrete-event simulation of multi-media telecommunication systems. In 2000, he was appointed by UNS, and he joined the MASCOTTE research group, a joint team of UNS, CNRS and INRIA. His current research interests

in discrete-event simulation are on methodology support, very large-scale networked systems, and wireless communication systems. His email address is olivier.dalle@inria.fr.