Design considerations for M S software

Olivier Dalle, Judicaël Ribault, Jan Himmelspach

Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, eds.

DESIGN CONSIDERATIONS FOR M&S SOFTWARE

Olivier Dalle
Judicaël Ribault

INRIA - CRISAM &
University of Nice Sophia Antipolis - I3S-UMR CNRS 6070
BP93 - 06903 Sophia Antipolis, France

Jan Himmelspach

University of Rostock
18059 Rostock, Germany

ABSTRACT

The development of M&S products often seems to be driven by need: people start coding because they are interested in either a concrete simulation study, or they are interested in a (single) research subject of M&S methodology. We claim that discussing, designing, developing, and comparing M&S products should be based on software engineering concepts. We shortly introduce some of these engineering concepts and discuss how these relate to the M&S domain. By describing two examples, OSA and JAMES II, we illustrate that reuse might play an important role in the development of high quality M&S products as the examples allow reuse on the level of models and scenarios, on the level of "simulation studies", of algorithms (e.g., reuse of event queues, random number generators), across hardware architectures / operating systems, and of analysis tools.

1 INTRODUCTION

In the last years, literature appeared criticizing the quality and validity of modeling and simulation (M&S) results (Pawlikowski, Jeong, and Lee 2002, Edmonds and Hales 2003, Kurkowski, Camp, and Colagrosso 2005). They mostly criticize that important details about the underlying setups which form the base for the results presented are not mentioned in the literature they revised (e.g., which random number generators used), and that often no information about the validity of the results is given (e.g., confidence intervals, hints to applied validation techniques).

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, Law 2007, 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. All processes in M&S can be seen as engineering processes, 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'll 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. Nevertheless many of the techniques hereby discussed can be used for all M&S engineering dimensions.

The question addressed in this paper is to identify which requirements a good M&S software product must fulfill, and how software engineering can help in fulfulling 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. We shortly discuss their use in M&S and finally illustrate the techniques and patterns using two examples of M&S software.

2 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. Later on, in the examples section 4 we further explain how these have been put in practise in two examples.

2.1 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.

2.1.1 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 process 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.

2.1.2 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).

2.2 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.

2.3 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.

2.4 Quality management

Quality is a multi-faceted, but widely used term (Himmelspach and Uhrmacher 2009). And although many different notions of quality exist, almost everyone agrees to the statement 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, Jeong, and Lee 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, ...).

2.5 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".

2.6 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 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 wide-spreaded 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 wide-spread 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 2009). 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.

3.1 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?

3.1.1 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.

3.1.2 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 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.

3.3 Reuse based designs: why and how

In (Sommerville 2007, p. 417) a list of benefits of software reuse is given which comprises increased dependability, reduced process risk, effective use of specialists, standards compliance, and accelerated development. We strongly believe that these hold for any of the three engineering dimensions in M&S as well as, in particular, for M&S software building. And we believe that the list of problems with reuse (Sommerville 2007, p. 418) comprising increased maintenance costs due to changes in reused closed source elements (does not apply in full extent to open source software in anyway), lack of tool support for development, not-invented-here syndrome (might be the largest problem, but should be superseded by work of a specialist, standards, and well documented test results), cost of creation and maintenance of a component library (should not be as worse as redoing everything from scratch, each time), and retrieval, understanding and adaptation of reusable components (e.g., components if realized as plug-ins can be more or less automatically found and facilitated by the system) are not as worse as the problems (non expert developments, quality of the implementations, ...) arising out of reimplementation from scratch.

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.

This leads to our first conclusion and open perspective for M&S: 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.

4 TWO EXAMPLES

There are already some existing open architectures which can be used to create new M&S software products. Two of them, OSA and JAMES II, are described in the following by aligning their design and development to the engineering techniques named above. Thereby we especially answer the following questions: How do the two correspond to the architectural designs given above? Why have we created the products as we did? What's the difference between the two approaches?

4.1 OSA

OSA (Open Simulation Architecture) (Dalle 2006, Dalle 2007) 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.

Architecture OSA has been created based on 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, Zeigler, and Wainer 2008). Fractal comes with an Architecture Description Language (ADL) called FractalADL, 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 have been designed for the particular needs of M&S (simulation life-cycle, instrumentation, event scheduling, and so on).

Using the overloading technique described above, combined with the use of aspect-oriented programming (Kiczales 1996), 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 (Ribault and Dalle 2008).

Going further in reuse, OSA aims at reusing any valuable piece of third party tools and not just models or scenarios. As a matter of facts, the parallel and distributed version of OSA uses the layering principle and delegates the transparent distribution of components and the deployment to two third-party contributed pieces, called FractalRMI (Flissi and Merle 2006, available at `http://fractal.ow2.org` (accessed 2009/07/15)) and Fractal Deployment Framework (FDF, available at `http://gforge.inria.fr/projects/fdf/` (accessed 2009/07/15)), respectively. More broadly we aim at delegating as many simulation concerns as possible to third party tools. This includes instrumentation, analysis and post-processing, or even simulation engines and models from others simulators. For example, for the instrumentation we have chosen COSMOS (Conan, Rouvoy, and Seinturier 2007) to aggregate the data. Although COSMOS allows us to aggregate data on the fly in a smart way, we must instrument some variables in models. To do this we use aspect-oriented programming to instrument the code at the byte code level, such that the original source code of models is not *polluted* by non-functional concerns (Dalle and Mrabet 2007).

OSA is also meant to become a *front-end / back-end* architecture, since we plan 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)).

Development The layers development follows a *component-based model*. At the beginning we started to build OSA by creating a simulation controller component inside the membrane of standard Fractal components, which allowed us to get our first simulation in a very short time. In order to easily manage multiple versions of the components and organize their dependencies, we eventually adopted the Apache Maven Project tools. Thereafter we started to integrate third-party tools for instrumenting and deploying the simulation. In order to ensure quality, one of the first tasks in the OSA project was the definition of a *coding style* manual which clearly defines the coding conventions. These conventions are automatically enforced by a *project template model* when working with Eclipse in combination with the Apache Maven project management system. *Unit-testing* is based on *junit tests*. Regarding future developments, we are currently working on reusing parts and tools from various simulators (e.g., Scave, the data analysis tool from Omnet++) and in a near future we hope to reuse simulations engines and models from well-known simulators.

Team OSA is a recent project initiated by a small group in academia. The development started as a single person project, followed by an engineer and a PhD student, thus following a *chief programmer* organization. Although the staff in charge of OSA is small, the reuse of third-party tools permits to involve punctually third-party contributors. This results in reciprocal benefits, since contributors find in OSA new opportunities to improve their existing works based on new realistic use cases and conversely, the OSA project gains new functionalities with each new third-party contribution. Ultimately, the kind of organization targeted is a typical *distributed open source project* accepting any third-party contributions, with a BDFL manager (Benevolent Dictator For Life) responsible for taking major orientations and design decisions.

Status Currently and despite the fact that there are currently no component libraries provided with OSA, it is possible to develop new models from scratch, to build simple or complex scenarios and experimental frames, and to configure and deploy simulation runs in centralized or distributed mode. All or parts can be shared in order to be reused, thanks to Maven on-line repositories.

Use OSA is used in several projects, such as the INRIA-funded "BROCCOLI" project or the ANR-funded SPREADS project. The BROCCOLI project involves three French academic research teams and its goal is to design a platform for describing, deploying, executing, observing, administrating, and reconfiguring large-scale component-based software architectures. The project aims in particular, but not exclusively, at building large scale discrete event simulation applications. Thanks to the OSA layered architecture, each team can focus on their particular area of expertise.

The SPREADS project involves five research teams from industry and academia. It aims at studying issues related to peer-to-peer based storage and data backup systems. A common case study is shared between these two projects. It consists in the modeling and simulation of a large scale peer-to-peer based data storage system: the solutions found in the BROCCOLI project are applied to solve the issues raised by the large-scale deployment and simulation of the system under study in the SPREADS project. OSA is an open source project, available at `http://osa.inria.fr/`.

4.2 JAMES II

JAMES II (JAva-based Multipurpose Environment for Simulation II) has been created with three user groups in mind: users which want to learn about M&S, users which want to apply M&S, and users which want to conduct methodological research in modeling or simulation. Thus the resulting product shall be usable for three different use cases as well as for many differently skilled users. All solutions developed can coexist and users do not need to get in touch with details nor with parts they are not interested in, but they can.

Architecture JAMES II has been created based on a *plug-in architecture* (Himmelspach and Uhrmacher 2007). 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 basically built by 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 datasinks, 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 the one hand (in the large) 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 in there allow to add front-end plug-ins for many back-end parts. On the other hand (in the small) JAMES II supports the differentiation between symbolic (front-) and executable models (back-end). Although JAMES II has been basically built using a plug-in based architecture it can be described best by using a *mixture of architectures*.

Development At the beginning we started to realize JAMES II by making use of *prototyping (vertical prototyping)*. This helped us in getting a lot of knowledge in a rather short time (the first prototype was realized in about two weeks), and we had immediately been able to produce very first insights. The version currently available differs from the prototypes we created over the years in many aspects: we had to learn which requirements are there (from different execution paradigms, modeling paradigms and formalisms), and what seems to be a good solution to these. In the last two years we switched over to an *evolutionary model* for the core of the product. We make a strict difference between the development of the core and the development of plug-ins: the core shall not have any dependencies to any other library, plug-ins might have. The plug-in development follows a *component based model* – here we reuse other plug-ins and third party products, and thus reduce development time and costs. This distinction shall help to embed JAMES II into any other application (no library conflict can occur by using the core – plug-ins can be replaced always, and thus you can avoid library conflicts easier as if you would have to align your product to the libraries used by the core), and the JAMES II core remains relatively lean. This means that we have / had to develop many existing things from scratch – but you do not need to use these at all – you can still use, e.g., Eclipse as environment, and JAMES II as back-end. *Quality* is a severe issue which we try to deal with as our capacities allow: At first the development has been accompanied by strict *coding conventions*, and a careful setup of compiler warnings in combination with the permant rule to get rid of all warnings. Recently we started to release the product, and from now on we are using a *build management system*. For many classes in JAMES II, especially for some commonly used plug-in types, we have *junit tests*, which are now automatically executed during each build. In addition we have an extra extension providing tests for the random number generators – because they can hardly be tested by usual junit tests.

Team JAMES II is being created in an academic environment. The team formation was a hierarchical *chief programmer* organization from the beginning on. Thus the team is mostly formed by people working on a thesis, and on students earning some money. Consequently the choice what to develop next depends on the skills of the people at hand – thus here we do not have mostly a need but a resource driven development. The time spans people are involved in the project varies but are most often relatively short. The range is typically from three months up to 2 years for most students, for phd candidates this ranges from 3 to around 5 years. If the latter started as students their overall contribution is typically very good. As always the skills of the team members differ significantly, but maybe its even worse in our academic setting: we are still educating the contributing team members. Thus most of the code had to be reviewed by more experienced team members later on. So far more than 30 people contributed to the project, among the contributions 2 finished (and 6 running) phd theses; 16 master theses; 14 bachelor theses (or equivalent contributions). A major problem in the academic setting is the consequent execution of a *project plan*: often you have to wait rather long until you find someone willing to contribute and having the appropriate skills. You simply cannot start searching for an expert and "buy him / her" – consequently the team size varied strongly over the years (from 2 to 15 people), and thus the progress made. Although the plug-in based approach helps new team members (they usually have only to write their code against one interface for which a unit test often exists), many team members needed a couple of weeks or months until they started to contribute. The different team sizes require different management strategies, and without a common coding style, the plug-in approach, and a repository based development it

would be hard to merge all the results. Altogether this makes time planning rather hard, and whether a milestone (e.g., as a result of a master thesis) will be reached or not is partially out of control.

Status Currently there are more than 400 plug-ins available for use in JAMES II. These include a variety of modeling alternatives, simulation algorithm alternatives for these, simulation experiment alternatives, data sinks, random number generators, event queue implementations and many more. Recently we started to integrate research results (software) from other research groups, e.g., for up to date visualization techniques.

Use The framework has already been used in different domains, among these experimental algorithmics, demographic predictions, systems biology, network protocol evaluation, and software testing. Currently steps are undertaken to use the framework for traffic simulation and economic predictions. Whatever the application was, large parts of JAMES II had been reusable, no need to develop the whole system from scratch arose – at most new plug-ins had to be added extending the framework with the functionality required. JAMES II is an open source project, and you can get it from http://www.jamesii.org.

4.3 A comparison

Both, OSA and JAMES II, have been created to be open architectures, both have been created based on the idea of exchangeable parts. Thus they shall be reusable for a variety of applications, from different domains, and they shall serve as base to integrate additional functionality.

Besides general similarities there are several differences between OSA and JAMES II. The most apparent one is that OSA started by making use of reuse: OSA is based on the fractal framework. JAMES II has been created from scratch, and thus for JAMES II everything contained in the fractal framework had to be recreated as soon as we needed it. A pretty good example for the difference is the fractal deployment framework: OSA does not need to worry about a distributed setup too much. You just have to define a single XML file containing the setup – for JAMES II we had to design, and realize the distributed infrastructure and we have to cope with communication issues on our own. The reason for redoing everything from scratch in JAMES II was and is that the core of JAMES II shall be reusable without any dependencies to any other products – something you cannot get for free. The underlying architectural concepts of OSA and JAMES II are different as well. OSA has been created based on a layered architecture whereby JAMES II is created based on a plug-in architecture. Nevertheless both products do not differ very much in the options they provide: The plug-in based architecture of JAMES II is partially substituted by the fractal component model, the distinction between a front-end and a back-end is available in both as well. In OSA currently mostly FractalADL is used for the front-end so far, in JAMES II the front-end metaphor is used for experiment and model definitions (here we already provide support for a variety of modeling languages) as well as for a general graphical user interfaces to the framework. Currently OSA is restricted to discrete event based simulations, the core of OSA contains different packages containing the basic event scheduling and a part of the fractal launcher which can read the initial events. The core of JAMES II contains much more, but less concrete – more, e.g., all the functionalities and services provided by the fractal environment for OSA, less, e.g., the core of JAMES II does not support any modeling paradigm nor any scheduling / synchronization mechanism. OSA only allows access to simulation services via a well-defined API. In the current version the classical view of process based simulation is facilitated, and thus models call simulation API methods. JAMES II enforces a very strict separation of concerns: models are not allowed to call methods in the simulation algorithm classes, access is always from the simulation algorithm to the model.

Nevertheless both systems allow to integrate parts of the other. OSA could be used to execute simulation jobs created by the experimental layer of JAMES II, and OSA could integrate, by exploiting the fractal design, plug-ins from JAMES II (as event queues, random number generators, ...).

5 CONCLUSION AND OUTLOOK

We have discussed the need to apply architectural design patterns for the software engineering process of M&S applications. They can help on many facets (discussion, description, development, maintenance, ...) and thus they may add to the overall quality of M&S results. All architectures can be applied for designing M&S tools – standalone or in mixed variants. Two M&S products, OSA and JAMES II, are described, discussed, and compared based on the architectural patterns. Both are flexible and extensible, and can be used as a base for more specialized M&S products. Thereby, they can significantly lower the overall development effort, which means that more time remains for other tasks, and thus they can add to the overall quality of M&S results produced based on these as well. Tools like these could help to move the whole field forward: by reducing the workload per idea, and thus by allowing to concentrate more on the actual subject of research, instead of spending a lot of time reimplementing the wheel.

Both M&S software descriptions (OSA and JAMES II) contain a paragraph on the respective development teams which make clear that a well-organized, and planned project management of large projects as the development of software for M&S is hard to do in academia: teams might be rather small, or of varying sizes, and the skills of team members may vary perceptibly. Consequently projects might run for quite a long time until the initial task can be done with a sufficient level of quality. Intensive reuse can help to overcome some of the related problems, and thus can help to focus on the real task, and not on the development of already existing material. Therefore we can only ask everyone to contribute to open approaches: if experts contribute reuse gets ever more fun, and the quality will most hopefully be significantly increased in the long run. However, all M&S products should be developed based on state of the art (software engineering) methods. To have well defined M&S products around, which are well documented, can help to create and to communicate credible M&S products.

In the future we will start working on the compatibility of OSA and JAMES II. Because we strongly believe that repeatable experiments across M&S products, reuse in and of these, and their interoperability, and thus the combination of work forces, are the only keys to achieve credible simulation results.

REFERENCES

Balci, O. 2003. Verification, validation, and certification of modeling and simulation applications. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, 150–158: Winter Simulation Conference.

Balci, O., and R. E. Nance. 1992. The simulation model development environment: an overview. In *WSC '92: Proceedings of the 24th conference on Winter simulation*, 726–736. New York, NY, USA: ACM.

Bruneton, E., T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani.. 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.

Conan, D., R. Rouvoy, and L. Seinturier. 2007, June. Scalable processing of context information with cosmos. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'07)*, Volume 4531 of *Lecture Notes in Computer Science*, 210–224: Springer.

Dalle, O. 2006, May. Osa: an open component-based architecture for discrete-event simulation. In *20th European Conference on Modeling and Simulation (ECMS)*, 253–259. Bonn, Germany.

Dalle, O. 2007, February. 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*, 213–218. Buenos Aires, AR.

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

Dalle, O., B. P. Zeigler, and G. A. Wainer. 2008, December. Extending DEVS to support multiple occurrence in component-based simulation. In *Proceedings of the 2008 Winter Simulation Conference*, ed. S. J. Mason, R. R. Hill, L. Moench, and O. Rose.

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

Edmonds, B., and D. Hales. 2003. Replication, replication and replication: Some hard lessons from model alignment. *J. Artificial Societies and Social Simulation* 6 (4).

Endres, A., and D. Rombach. 2003. *A handbook of software and systems engineering*. Essex, England: Pearson Education Ltd.

Flissi, A., and P. Merle. 2006. A generic deployment framework for grid computing and distributed applications. *LNCS* 4276:1402.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

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

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

Himmelspach, J. 2009, September. 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.

Himmelspach, J., and A. M. Uhrmacher. 2007, March. Plug'n simulate. In *ANSS '07: Proceedings of the 40th Annual Simulation Symposium*, 137–143. Washington, DC, USA: IEEE Computer Society.

Himmelspach, J., and A. M. Uhrmacher. 2009, June. What contributes to the quality of simulation results? In *Proceedings of the 2009 INFORMS Simulation Society Research Workshop*, ed. L. H. Lee, M. E. Kuhl, J. W. Fowler, and S. Robinson.

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

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

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

Kurkowski, S., T. Camp, and M. Colagrosso. 2005. Manet simulation studies: the incredibles. *SIGMOBILE Mob. Comput. Commun. Rev.* 9 (4): 50–61.

Law, A. M. 2007. *Simulation modeling and analysis*. 4 ed. McGraw-Hill International.

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

Madsen, K. 2003. Five years of framework building: lessons learned. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 345–352: ACM Press.

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

Ribault, J., and O. Dalle. 2008. Enabling advanced simulation scenarios with new software engineering techniques. In *20th European Modeling and Simulation Symposium (EMSS 2008)*. Briatico, Italy.

Sargent, R. G. 2008. Verification and validation of simulation models. In *Proc. of the 39th WSC*, ed. S. Mason, R. Hill, L. Moench, and O. Rose, 157–169.

Sommerville, I. 2007. *Software engineering*. 8 ed. Addison-Wesley.

Troitzsch, K. G. 2004. Validating simulation models. In *18th European Simulation Multiconference. Networked Simulations and Simulation Networks*, ed. G. Horton, 265–270. The Society for Modeling and Simulation International: SCS Publishing House.

Van Waveren, R. H., S. Groot, H. Scholten, F. Van Geer, H. Wösten, R. Koeze, and J. Noort. 2000. *Good modelling practice handbook*. Utrecht, RWS-RIZA, Lelystad, The Netherlands: STOWA.

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

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

AUTHOR BIOGRAPHIES

JUDICAEL RIBAULT is a doctoral candidate at University of Nice-Sophia Antipolis (UNS). He received his MS degree in Computer Science at UNS. He is doing his research within the MASCOTTE common project-team of the I3S-UNSA/CNRS Laboratory and INRIA, in Sophia Antipolis. His research interests are on Component-based software engineering and simulation. His web page an be found at <http://www-sop.inria.fr/members/Judicael.Ribault/>.

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

JAN HIMMELSPACH is a post doc in the Computer Science Department at the University of Rostock. He received his doctorate in Computer Science from the University of Rostock. His research interest is on software engineering for modeling and simulation, credibility of modeling and simulation, and on efficient modeling and simulation solutions. His web-page can be found at <wwwmosi.informatik.uni-rostock.de/mosi/Members/jh194>.