Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

TUTORIAL ON BUILDING M&S SOFTWARE BASED ON REUSE

Jan Himmelspach

Albert Einstein Str. 22
University of Rostock
18059 Rostock, GERMANY

ABSTRACT

The development of software for modeling and simulation is still a common step in the course of projects. Thereby any software development is error prone and expensive and it is very likely that the software produced contains flaws. This tutorial will show which techniques are needed in modeling and simulation software independent from application domains and model description means and how reuse and the use of state of the art tools can improve the software production process. The tutorial is based on our experiences made on developing and using JAMES II, a flexible framework created for building specialized M&S software products, for research on modeling and simulation, and for applying modeling and simulation.

1 INTRODUCTION

Modeling and simulation (M&S) is in a credibility crisis (Pawlikowski, Jeong, and Lee 2002; Wilson 2006; Merali 2010). Among the contributing problems the software (quality) plays a central role (Wilson 2006; Merali 2010). As a main reason for low software quality the development as such has been identified.

Software development is a complex process, the importance of which seems to be often underestimated and which should be done by trained persons which know about the software engineering techniques to be applied (Wilson 2012). To improve software quality in general well defined development processes (Sommerville 2007; Dalle, Ribault, and Himmelspach 2009) are mandatory. This includes a careful testing of the diverse parts of the software developed, a process which consumes much time (test development and test execution after changes). In M&S software development implies the need for implementations of a number of techniques (Himmelspach 2009) which partially require advanced knowledge in mathematics and selected computer science techniques (e.g., numerical stability, numerics and hardware, random number generation, databases, parallel computation, ...). Consequently authors of such software should not only be trained in software development but also in those disciplines they need techniques from. Not in all settings where software is developed from scratch developers know from the beginning on which features they will have to add over time. This might lead to the situation that the overall effort for an implementation from scratch is underestimated (e.g., one might think that there is no need for supporting parallel computations in the beginning but learn later (if a model is created) that it will take too long until the results are computed sequentially). The maintenance aspect adds a further dimension to the problem of software quality. Software maintenance is a costly (approx 2/3 of the overall cost/time is said to be needed for maintenance (Sommerville 2007)) but required process. In an academic world where people usually develop software to earn a degree software maintenance and continued development is not as frequent as it should be. In summary

- ensuring software quality imposes much work (design, testing, documentation),
- the implementation of M&S techniques requires knowledge and can be troublesome,
- software maintenance is mandatory but expensive, and
- one might have to do more work in the end than it had been anticipated at the beginning.

978-1-4673-4781-5/12/$31.00 ©2012 IEEE

Minar, Burkhart, Langton, and Askenazi (1996) wrote more than a decade ago: "Once a field begins to mature, collaborations between scientists and engineers lead to the development of standardized, reliable equipment (e.g., commercially produced microscopes or centrifuges), thereby allowing scientists to focus on research rather than on tool building. The use of standardized scientific apparatus is not only a convenience: it allows one to "divide through" by the common equipment, thereby aiding the production of repeatable, comparable research results." This would help to resolve the issues we explained before but means that we are in the need of more reuse.

Examples like SWARM (Minar et al. 1996) already show that a continued development can be very fruitful in mid term to long term but such examples have the problem that they still have a limited applicability and thus a limited reusability: They are created for certain modeling domains / applications and are thus not generally reusable.

Several approaches have been undertaken in scientific computing to provide reusable functionality. A number of libraries, e.g., SSJ (L'Ecuyer and Buist 2005) and Colt (CERN 2012), have been developed for scientific computing, but as they provide only limited support (see Section 3) their usage is not sufficient to overcome all of the problems associated with creating a software for M&S mentioned before. μsik (Perumalla 2005), based on a micro kernel, offers reuse for the development of a special group of computation algorithms for one modeling means (logical processes) and is thus too restricted as well. More work in the M&S community deals with reusable model (components). E.g., in SimBeans (Praehofer, Sametinger, and Stritzinger 2001) the Java bean component model is facilitated, the HLA (Kuhl, Weatherly, and Dahmann 2000) allows to reuse models (computed by an extra piece of software), COMO (Röhl and Uhrmacher 2006) and the approach by Szabo and Teo (2011) focus on composing models by well defined specialized languages and mechanisms. Valentin, Verbraeck, and Sol (2003) examined the impact of using building bricks to create models, to use model libraries is common in software like MatLab/SimuLink or those based on MODELICA (Elmqvist, Mattsson, and Otter 1998). Others allow to combine models created in different formalisms / languages (and thus to reuse those), e.g., Möbius (Clark et al. 2001). Herein we concentrate on reuse for creating MSSP (Modeling and Simulation Software Products) and not on reuse for creating models.

One of the aspects making software development in M&S more complicated can be tracked down to terminology and associated with that to different view points. Many developers in M&S do not seem to know whether there software is a framework, a library, an architecture, a platform, a tool, ... as they use all these terms arbitrarily to describe their software making it hard to decide whether the software at hand is reusable for the current purpose. Further on not all approaches clearly separate between model, its computation, and the experiment definition. As separations of concerns is the key concept for intensive reuse any unnecessary mixture of concepts hampers reuse.

Herein we discuss how software products for M&S can be created to allow maximum reuse and how such products can be built based on reuse, i.e., we do not deal with building different models or experiments.

In the following Section we give a condensed overview of reuse techniques for software developments. Section 3 presents M&S techniques reusable in principle. Section 4 will show how a plug-in based framework can be used to create new M&S software products and how it can support research on M&S methods.

2 REUSE TECHNIQUES IN SOFTWARE DEVELOPMENT

Reuse and separation of concerns are inseparably intertwined. A consequent separation of concerns (SOC) allows to identify elements which can be extracted from a complex piece of code and to hide the functionality behind an interface. This reduces on a first glance code complexity and on a second view it allows to come up with alternative solutions for the code extracted. One approach to build systems based on the SOC and reuse is to use components (Szyperski 2002). A step beyond the pure SOC is the identification of patterns, e.g., Gamma, Helm, Johnson, and Vlissides 1995, which can further help to create software based on reuse and to design software products, which lend themselves for reuse.

Some languages (e.g., Delphi) provide building bricks for software based on the idea of visual components. In such languages at least the user interface can be created by dragging, dropping, and connecting reusable visual elements.

A common way to provide reusable functionality are **libraries** which are frequently used (e.g., dynamic link libraries). Languages like C and Pascal support this view on the language level by allowing to include files / units providing such functionality natively.

Frameworks extend the idea of a library by providing control logics, combining the functionality provided internally, to further ease the creation of applications for the domain the framework has been created for (Madsen 2003). An example is the Eclipse framework which eases the creation of integrated development environments for programming languages.

Workbenches allow to combine (e.g., by using pipes and filters) diverse tools (software dedicated to a single purpose, e.g., a compiler) to build more complex functionality.

Further on a number of techniques have been developed to provide access to reusable functionality. Among those the idea of services is of central interest as it maps well to recent developments like cloud computing or multi core computers. Services encapsulate reusable functionality which can be acquired from the outside through well defined interfaces.

This short overview shows that reuse is nothing new and that different techniques have been developed to support reuse. In the following two sections the two more classical approaches are discussed in more detail followed by a section on the plug-in concept which allows to create more flexible applications.

2.1 Reuse Provided by Libraries

Libraries provide a set of methods (data structures and algorithms) which can be reused in software. Such methods (e.g., data structures like queues, algorithms like sorting methods, etc.) can be integrated in new software as if it would have been created directly for the software. The advantage is that one usually do not need to worry about the actual implementation — you can "simply reuse" the functionality available. Programming languages such as Java ship with a set of such libraries, e.g., providing data structures like lists etc. The integration of functionality from a library is thereby usually hard coded: even if alternative methods implement the same interface (e.g., List in Java) one usually writes declarations like

```
List myList = new ArrayList();
```

This makes it rather difficult to exchange the list implementation used later on (as the code needs to be modified). However, the coding effort is significantly reduced, as well as the number of potential bugs in the new software created.

2.2 Reuse Provided by Frameworks

The intention of a framework is to ease the development of software products for the domain they are created for. A framework can be used as a library but it provides even more to further ease the creation of software. Frameworks contain more complex processes which represent typical parts of software from the domain which can be reused out of the box and which combine the library functionality shipping with the framework.

For example in a framework for M&S software we could have an `Experiment` class which can be used to execute experiments.

```
// ... your code

Experiment myExperiment = new Experiment(myModel);
// e.g., define which configurations to be executed
setupExperimentParameters (myExperiment);

// e.g., attach observers to visualize the progress of the experiment in your GUI
```

```
attachObserver(myExperiment);

myExperiment.execute();
```

This class may internally use lots of the functionality provided by the framework (like model reading, parameter computations, model computation algorithms, etc ...), however, that's completely hidden from a user. Further on, as being part of a framework, such classes like the Experiment class used above typically provide the ability to control (start, stop, observe, etc ...) the processes started, e.g., using an observer mechanisms which allows to embed such a piece of functionality seamlessly and with less coding efforts into specialized software products.

A framework does provide more reusable elements and thus further reduces the coding effort and thus the number of potential bugs in a new software as well. However, still, all the connections between the diverse elements in the software are hard coded, i.e., you need to recompile your application to exchange elements.

2.3 Reuse and Plug-ins

Plug-ins, i.e., the ability to add functionality to an installed software package without the need to recompile parts of the software, are a commonly used concept (e.g., web browser, office software, Eclipse).

Plug-ins (aka extensions) are of a type (plug-in type) and are provided for an extension point. Software products may only provide one extension point or they may differentiate a number of plug-in types. For each of the types interfaces may exist which plug-ins have to implement / via those plug-ins can access the software they are installed in. Each plug-in based software requires a plug-in management component where plug-ins are registered and where they can be retrieved from.

For example (assuming a plug-in type *List*):

```
List myList = PluginManagement.getInstance (ListImplementation, parameters);
```

This means that the decision which implementation to be used is moved to the management class which can make this decision based on the plug-ins currently available. Users can install plug-ins developed by third parties which make such systems adaptable, flexible, and, if the plug-in concept is consequently applied, scalable.

3 REUSABLE ELEMENTS OF M&S SOFTWARE

Languages like Delphi provide a library of reusable elements (e.g., buttons, text views) which ease the creation of user interfaces of applications. But what are reusable elements for M&S software? The following table gives such reusable elements (see also Himmelspach 2009) and shows which of those are available in a programming language, two libraries, and one framework usable for creating MSSP.

Reusable element	**Programming language**	**Libraries**		**Framework**
	Java	SSJ	Colt	JAMES II
event queues	yes	yes	no	yes
random number generators	yes	yes	yes	yes
random distributions	no	yes	yes	yes
random seeds generators	no	yes	yes	yes
replication criteria	no	no	no	yes
data sinks	no	no	no	yes
computation stop criteria	no	no	no	yes
optimization algorithms	no	no	no	yes
statistical methods	no	yes	yes	yes
graphical output (charts,...)	no	yes	yes	yes
logging support	yes	no	no	yes
GUI framework	basic	no	no	special (use optional)
experiment control	no	no	no	yes
experiment persistence	no	no	no	yes
parallel configurations	no	no	no	yes
parallel replications	no	no	no	yes
computation server support	no	no	no	yes
partitioning algorithms	no	no	no	yes
resilience	no	no	no	yes
automated selection of algorithms	no	no	no	yes
benchmarks	no	no	no	yes
reporting	no	no	no	yes
dynamic calculation trees	no	no	no	yes
...				

This comparison only includes Java related approaches but the general relations between the features provided by a general purpose programming and those delivered by dedicated libraries and frameworks hold always true.

All the elements given here are independent from concrete applications and most are independent from concrete modeling means as well.

3.1 Requirements of Reusable M&S Software

Due to the number of application domains, modeling formalisms, experiments, and hardware infrastructures a reusable MSSP needs to be as flexible as possible. Such a software should not be created with any concrete of these dimensions in mind. Further on the software reused should have been developed based on a well-defined architecture, developed in a well-defined setup, carefully tested, and documented (Dalle, Ribault, and Himmelspach 2009). The software reused should still be maintained and open (source code available) as otherwise time invested might be lost as soon as a problem has been identified in the reused project.

4 A REUSABLE SOFTWARE FOR THE DEVELOPMENT OF SOFTWARE FOR MODELING AND SIMULATION: PLUG'N SIMULATE AND JAMES II

Before we motivated reuse, we have shown what could be done to achieve reusable software, and we have shown what could be provided as reusable bricks for building M&S software. These insights lead to the development of an architecture named "plug'n simulate" (Himmelspach and Uhrmacher 2007b; Dalle, Ribault, and Himmelspach 2009). Thus JAMES II is based on extension points and extensions which is

the base for adding functionality to the software by everyone. Further on JAMES II has been created as a framework to provide at a maximum degree support for those developing specialized M&S software.

This makes JAMES II one example for reusable software which we will use from now on to show how MSSP can be built based on reuse. Further on we will show how research can be facilitated by such a software product as well.

JAMES II is available as open source and it has been created using software development means like repositories, unit testing, code analysis, continuous integration, and more to increase the trust in the product. It is distributed under an open source license which allows to reuse JAMES II, to extend JAMES II and to distribute your software built based on JAMES II.

4.1 Reusable Code

JAMES II has been created for reuse and developments for JAMES II are either developments to be added to the core (which has no further dependencies) or they are added to sub projects (e.g., support for a process algebra modeling formalism).

The core of JAMES II comprises more than 38,000 logical lines of code in $\approx$ 2000 classes grouped in $\approx$ 340 packages. These lines have been developed in between 2003 and 2012 with a varying number of contributors. Most of the code has been supplied by three authors which are in the project for 9, 7, and 4 years. To illustrate the efforts required we use in addition a less subjective measurement, COCOMO II™, with all options left on "nominal", besides the personnel expert levels trimmed to represent a "perfect" setting (assuming that the programmer is a perfect analyst, programmer, has a very high application, platform, and language/toolset experience, and is fully available for the project), and highly using supporting software tools. Based on the lines of code of the core we computed that more than 31 person months are needed to create this piece of software. This time span has to be multiplied with 5 by just lowering the personnel expertise and tool usage levels to mean values (which holds true if you assume that an author of such a software might not be a perfectly trained and experienced software engineer). And this is only the code available in the current release of the core, not including concrete modeling means, models or experiments. If you don't need 75% of this code you would still need, according to the optimal settings, 7 person months.

It is important to note that not all code which might be required to build a MSSP is placed in the core. For example data collection means (DB access, CSV files, etc.) or extended event queues are not included.

Even if you argue that this cost model is far from being perfect and that you need less than 25% of the functionality provided, it may help to understand that already building the basic functionality can consume a considerable amount of time which can be totally saved by reuse.

4.2 Building M&S Software Based on Reuse

There are several alternatives how JAMES II can be used to assist you in creating your specialized M&S software. We believe that it does not make much sense that everyone interested in creating a specialized M&S software should implement all the methods he needs in the end from scratch: why not reuse functionality required and only implement the functionality which is missing or of which you don't like the implementations of? That's what JAMES II has been created for: based on a pure plug-in architecture (plug'n simulate). In the following we describe the alternative reuse options and shortly give some pros and cons for each.

4.2.1 Using JAMES II as Library (simple)

JAMES II can be used as a library. Therefore you can simply use the interfaces and implementations provided by JAMES II, e.g., instead of those provided by Java. For example, you will no longer use the Queue interface of Java but the `IEventQueue` interface of JAMES II.

The small example used here is basically the classical Hold operation (see Algorithm 1). The computation will last until there are no more events to be processed or until we reached a pre-defined point in time.

Algorithm 1 Model computation based on the hold loop using a selected implementation provided by JAMES II. Stop condition is a fixed time value.

```
class MyProcessor extends Runnable {

double stopTime;
boolean stopping = false;

/*
 * Method to execute a model computation.
 */
void run () {
  IEventQueue queue = new PriorityEventQueue();
  double time = 0.;
  // fill the queue
  init();
  while (!queue.isEmpty() && (Double.compare (time,stopTime) < 0) && !stopping) {
    nextStep();
  }
}

void nextStep() {
  Entry<Event, Double> event = queue.dequeue();
  //advance the time
  time = event.getTime();
  //execute the event
  List<Entry<Event, Double>> newEvents = model.executeEvent (event);
  //enqueue new events
  for (Event newEvent : newEvents) {
    queue.enqueue (newEvent);
  }
}

... //e.g., methods to stop execution
```

Pros

- minor changes to existing code (for integration and removal)
- your user interface remains untouched

Cons

- less flexibility
- manual adaptation of code needed to adapt computation to problem characteristics (e.g., to change the event queue implementation because another one could perform better)
- many features of JAMES II remain unusable (experimentation layer, automatic configuration, etc)
- only implementations known during development time can be used

Some of the implementations provided within JAMES II are classical methods in a library (e.g., parts of the math, the computation tree, graph classes, etc). They have to be always reused in this manner.

4.2.2 Using JAMES II as Library (extended)

JAMES II is based on plug-ins and thus during application development we do not necessarily know which event queue implementations (or other plug-ins for different plug-in types) will be available. On startup JAMES II will search for plug-in types and plug-ins and register those in a central registry. This registry can be queried to get a factory which with we can create the functionality required.

For example, we can change the example given above to use more interfaces and factories shipping with JAMES II (see Algorithm 2). The event queue is now dynamically retrieved from the registry. As

parameters the model and the computation algorithm can be passed to allow the registry to select a "good" one. The `stopPolicy` replaces in the example the fixed time based cancellation by an arbitrary one. Here we assume that it has been created beforehand.

Algorithm 2 Model computation using the registry of JAMES II to retrieve an arbitrary instance of the functionality (event queue) required. Arbitrary stop condition can be retrieved from the registry of JAMES II by instance creating class.

```
class MyProcessor extends Runnable {

IComputationTaskStopPolicy stopPolicy;
boolean stopping = false;
IEventQueue queue;

public MyProcessor (IModel model, IComputationTaskStopPolicy stopPolicy) {
  super(model);
  this.stopPolicy = stopPolicy;
  queue = SimSystem.getRegistry().getFactory(AbstractEventQueueFactory.class, parameter).create(
      parameters);
  // fill the queue
  init();
}

/*
 * Method to execute a model computation.
 */
void run () {
  double time = 0.;
  while (!queue.isEmpty() && !stopPolicy.hasReachedEnd() && !stopping) {
    nextStep();
  }
}

void nextStep() {
  Entry<Event, Double> event = queue.dequeue();
  //advance the time
  time = event.getTime();
  //execute the event
  List<Entry<Event, Double>> newEvents = model.executeEvent (event);
  //enqueue new events
  for (Event newEvent : newEvents) {
    queue.enqueue (newEvent);
  }
}

... //e.g., methods to change stopping condition; to stop computation etc ...
```

Pros

- still minor changes to existing code (for integration and removal)
- more flexibility than in the simple case
- implementations unknown during development time can be used
- your user interface remains untouched

Cons

- many features of JAMES II remain unusable (experimentation layer, automatic configuration, ...)

4.2.3 Using JAMES II as Framework (backend)

JAMES II has been designed as a framework and thus this type of reuse offers the most for your specialized M&S software. Still your software stays your software, i.e., your software XYZ is still XYZ if based on JAMES II. If JAMES II is used as backend the registry becomes the central resource for functionality in your application. Whenever one needs a special functionality, e.g., a special distribution, the registry can be queried for an implementation. Further on experimentation will be handled by the experimentation layer of JAMES II. The computation algorithm/simulator and even its configuration will be automatically determined by JAMES II in the background. To provide these implementations to JAMES II they may need to implement special interfaces (to fulfill the requirements of the plug-in types). For example, the `IProcessor` interface needs to be implemented by your computation algorithms and models need to implement the `IModel` interface. Using functionality like event queues is thereby still like given in the example before. Everything else you might need for doing experiment should be handled by the experimentation layer and in case that the functionality you need does not exist (e.g., a special replication criteria) you can add this as a new plug-in.

Algorithm 3 Model computation using extended functionality in JAMES II. Implementation effort is reduced on the code required for the computation of a single step. Stop condition creation, getting event queue, loop control etc. is handled by the framework, respectively by the factory classes for plug-in instances.

```
class MyProcessor extends RunnableProcessor {

IEventQueue queue;

public MyProcessor (IModel model, IEventQueue queue) {
  super(model);
  this.queue = queue;
}

/*
 * Method to execute a step in a model computation.
 */
void nextStep () {
  Entry<Event, Double> event = queue.dequeue();
  //advance the time
  setTime (event.getTime());
  //execute the event
  List<Entry<Event, Double>> newEvents = model.executeEvent (event);
  //enqueue new events
  for (Entry<Event, Double> newEvent : newEvents) {
    queue.enqueue (newEvent);
  }
}

}
```

The small examples used before has been altered (see Algorithm 3) so that JAMES II takes over more responsibility. Dealing with stop policies and looping is controlled by JAMES II now. Further on additional features like manual cancellation and pausing are added automatically. By inheriting from the `RunnableProcessor` the `IProcessor` interface is implemented automatically.

Pros

- all features of JAMES II are usable (experimentation layer, automatic configuration, ...)
- maximal flexibility
- implementations unknown during development time can be used
- actively maintained frameworks like JAMES II are being constantly under development and very likely to be improved in the future: if your application is based on such a framework it automatically

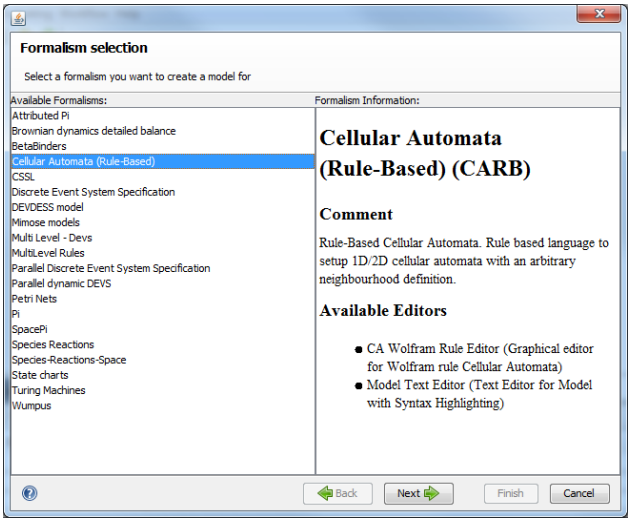

Figure 1: JAMES II can be used as an environment which can be extended via plug-ins by modeling means and any number of editors dedicated to these. Other functionality, e.g., for setting up an experiment, can be reused for any of those.

inherits these improvements, e.g., a bug fix in or new implementations of an event queue will be available to your application, almost for free
- your user interface remains untouched
- users can concentrate on their research topic (e.g., novel computation algorithms, novel modeling means) and do not need to worry about the experimentation layer etc ...

Cons

- major changes to existing code (for integration and removal)

4.2.4 Using JAMES II as Environment

As not everyone interested in M&S likes the idea to create a complete application from scratch we created a second framework in our framework: a framework for graphical user interfaces. Based on this framework we created a simplistic environment which can be extended using the JAMES II plug-in system as well. You can add editors for your modeling language and reuse the experiment setup dialogues etc (see Figure 1).

This case can be combined with the framework use case in case that a new modeling formalism shall be added but no specialized GUI shall be created. The pros and cons are those mentioned above (using as framework) plus

Pros

- you get pre-fabricated GUI elements

Cons

- the interface of your application will most likely need to be changed as well if you already have an implementation

4.2.5 Using JAMES II as Workbench

JAMES II can be used to control any external software by using the experimentation layer as well. This means that instead of reusing internal functions and plug-ins the software (and all models realized for etc) is integrated as a black box. JAMES II than simply calls this software during the course of an experiment.
Pros

- no changes to your software needed (as long as it does not prevent parallelism, i.e., being called more than once)
- exploits partial features of the experimentation layer

Cons

- techniques/methods provided by JAMES II are not used (e.g., event queues, etc.)
- experiment control techniques (like stop policies) cannot be used
- JAMES II does not know what has been used in the software called (e.g., cannot control random number generation)
- data conversion might be troublesome (parameters needs to be transferred to the external application; data needs to be transferred from the external application to JAMES II)

4.3 Research for M&S Based on Reuse

Above we have shown that building software based on reuse can reduce the amount of code to be written and that one can inherit functionality which otherwise would have to be coded explicitly. Herein we will illustrate how code reuse can support doing (practical) research: often at least prototypes have to be developed to illustrate that a theoretical approach works in reality.

4.3.1 Developing New Modeling Means

For M&S many languages / modeling formalisms have been (Clema 1978) and are still being developed, e.g., Uhrmacher et al. 2008. In addition to the formal development of the language (syntax and semantics) each language development is usually accompanied by a (prototypical) implementation. For such an implementation you need at least

- classes/methods/data structures to represent instances of the model,
- methods to compute the model (aka simulators), and
- a mechanism to setup a simple experiment with example models.

However, this minimal set does not make the implementation necessarily reusable, neither by the author (if the experiments to be done leave the level of simple examples) nor by others for any experiments they might be interested in.

For example, Attributed Pi, a formalism created in a phd thesis (John 2011), has been realized with approx. 2300 logical lines of code based on JAMES II. According to the aforementioned cost model this means approx. 7 person months (leaving everything (complexity, experience, etc.) on the mean values). Thereby this implementation was done based on an implementation available for Stochastic Pi which reduced the amount of code to be written. Thus reuse of available implementations to build the new modeling means on plus the absence of the need to create the surrounding elements (experiment control etc.) leaves time in a thesis to work on further projects (languages, cooperations, etc ...).

Further on realizing the mandatory set is further simplified by the methods and data structures provided by the framework (e.g., random numbers, distributions, event queues) and by enforcing, from the beginning on, a strict separation of concerns for the new modeling means (e.g., dedicated parser to build instances of the model from a textual description, separation of model and computation classes).

4.3.2 Developing New and Experimentation with Computation Means

Developing alternative computation schema for the computation of models has a long history in M&S. Among the reasons are the computational requirements imposed by the single computation of a model, those imposed by a set of such computations, and new hardware developments. However, the evaluation of such means usually implies that one has to realize

- classes/methods/data structures to represent instances of the model,
- the new computation means, and
- *n* computation means the new one shall be compared with.

This rather classical procedure may introduce a number of pitfalls (Johnson 2002) which can be partially avoided if such an endeavor is made based on a framework like JAMES II. At first utility methods / data structures reusable in the development of new computation means might be better evaluated and tested than those "quickly" implemented to test the new idea. A general help is the separation of concerns enforced by the framework as it helps to concentrate on the algorithm and as it supports the identification of exchangeable parts of an algorithm (e.g., random number generation, event queues). If the model means, the algorithm shall be developed for, has already been realized only the new algorithm has to be added, which can then be compared to the alternatives already available. This avoids biased implementations of the model (e.g., using data structures supporting a concrete algorithm, although this can still be done thanks to the fact that computation algorithms only work on interfaces of models in JAMES II, but then the old algorithms can still be used in the new comparison) as well as biased implementations of the algorithms the new one shall be compared with. Further on a suite of test models might already exist which does not only speed up the overall process: it adds a further level of fairness assuming that test models are around which ease the computation of a model by selected algorithms and not only the computation of the newly developed one.

By realizing a new algorithm based on such a framework you end up with the chance to use the new algorithm, without any additional efforts, in full grown experiments. Last but not least realizing such an algorithm based on such a framework helps preventing the situation that the new algorithm developed, which might be better than others, is rarely used at all: it can be deployed and easily be used by all those using the modeling means it has been created for. There are a number of examples where JAMES II has already been used for experiments with algorithms, e.g., (Himmelspach and Uhrmacher 2007a; Rybacki, Himmelspach, and Uhrmacher 2009) and where based on the framework the experimentation methodology for experiment with algorithms has been advanced (Ewald, Himmelspach, Jeschke, Leye, and Uhrmacher 2009).

4.3.3 Developing New Techniques for Experiments

Besides models and computations means experiments in M&S might require further techniques (Himmelspach 2009; Ewald et al. 2010). Among those techniques for improving the scalability of the software, for optimization, validation, identification of patterns, warm up phases, computation stopping conditions, estimating the number replications, experiment design techniques, and so on. To experiment with any of these it is at least helpful if they can be evaluated based on a real experiment (with a real model). Without using a framework like JAMES II this means that aspects like experiment control, model classes, computation algorithm, etc. need to be implemented to test the new development. Adding them to an existing system allows a developer to concentrate solely on the new technique.

As one example the selection of "best performing" combination of algorithms to compute a model can be seen (Ewald, Himmelspach, and Uhrmacher 2008). Such a development is only possible if a portfolio of alternatives exist (algorithm combinations for different models per modeling means). Another example is the idea to control/steer simulation studies by using workflow technology (Rybacki, Himmelspach, Seib, and Uhrmacher 2010). Here the internal (hard coded) workflows of the framework are replaced by

exchangeable workflow definitions. Both examples can only be realized in mid to long term projects based on a well defined design and high reuse.

5 CONCLUSIONS

Due to the fact that software for M&S requires not only efficient and error free implementations of well known methods and data structures but also implementations of rather specific solutions (e.g., random number generators) implementations from scratch should be avoided as they are most likely very error prone. Reuse can help here as it avoids the re-implementation of such elements (if based on libraries and frameworks) and if reuse is based on frameworks even more complex workflows in software can be reused, further reducing efforts required and potential problems. Reuse requires a design based on the separations of concerns paradigm and implies the need to search for reusable elements and to establish trust in these. Our experience shows that frameworks created to ease the building of specialized M&S software, like JAMES II, can significantly reduce the overall workload, can speed up the development process of special solutions (e.g., add your own modeling formalism / computation algorithm) and immediately start with experiments (using optimization algorithms, parallel computation, etc.), and that they get better and better over the time as a whole interest group supports the maintenance process.

Although reuse helps to overcome quality issues reuse is not the sole key towards high quality code. Code development should nowadays be done supported by software development tools beyond the editor you are typing your code in. Tools created for supporting versioning and collaborative work, unit testing (continuously), static code analysis, and bug tracking should be used to further improve the quality of the software used for experiments in the computational sciences. Our experience shows that these tools, once installed, ease the development of software — for one person as well as for many persons projects.

ACKNOWLEDGMENTS

The current version of JAMES II has been created based on the work realized by more than 50 supporters. Parts of the project have been funded by the DFG (German research foundation) in the project COSA.

REFERENCES

CERN 2012. “Colt”. Accessed June 1, 2012. http://acs.lbl.gov/software/colt/.

Clark, G., T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. 2001. “The Möbius Modeling Tool”. In *9th international Workshop on Petri Nets and Performance Models (PNPM’01)*, edited by B. Haverkort and R. German, 241–250: IEEE.

Clema, J. K. 1978. “General purpose tools for system simulation”. In *ANSS ’78: Proceedings of the 11th annual symposium on Simulation*, edited by P. N. Adams, E. W. Hawes, Jr., and R. A. Pierce, 37–60. Piscataway, NJ, USA: IEEE Press.

Dalle, O., J. Ribault, and J. Himmelspach. 2009, December. “Design considerations for m&s software”. In *Proceedings of the 2009 Winter Simulation Conference*, edited by M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 944–955. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Elmqvist, H., S. E. Mattsson, and M. Otter. 1998, June. “Modelica - The New Object-Oriented Modeling Language.”. In *The 12th European Simulation Multiconference, ESM’98*, edited by R. N. Zobel and D. P. F. Möller, 127–131. Manchester, UK.

Ewald, R., J. Himmelspach, M. Jeschke, S. Leye, and A. Uhrmacher. 2009, October. “Performance Issues in Evaluating Models and Designing Simulation Algorithms”. In *Proceedings of the 1st HiBi conference*, edited by M. Valero and T. Mazza, 71–80. Piscataway, NJ: The Institute of Electrical and Electronics Engineers, Inc.

Ewald, R., J. Himmelspach, M. Jeschke, S. Leye, and A. M. Uhrmacher. 2010, January. "Flexible Experimentation in the Modeling and Simulation Framework JAMES II – Implications for Computational Systems Biology". *Briefings in Bioinformatics* 11 (3): 290–300.

Ewald, R., J. Himmelspach, and A. M. Uhrmacher. 2008, June. "An Algorithm Selection Approach for Simulation Systems". In *22nd Workshop on Principles of Advanced and Distributed Simulation*, edited by J. Liu and F. Quaglia, 91–98. Piscataway, NJ: The Institute of Electrical and Electronics Engineers, Inc.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, USA.

Himmelspach, J. 2009, September. "Toward a Collection of Principles, Techniques, and Elements of Simulation Tools". In *First International Conference on Advances in System Simulation*, edited by W. E. Biles, K. Hawick, and L. Mönch, 56–61. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc.

Himmelspach, J., and A. M. Uhrmacher. 2007a, March. "The event queue problem and PDEVS". In *Proceedings of the DEVS Integrated M&S Symposium*, edited by J. Nutaro and X. Hu, 257–264. Norfolk, VA: SCS.

Himmelspach, J., and A. M. Uhrmacher. 2007b, March. "Plug'n simulate". In *ANSS '07: Proceedings of the 40th Annual Simulation Symposium*, edited by H. Karatza and T. F. Znati, 137–143. Washington, DC, USA: The Institute of Electrical and Electronics Engineers, Inc.

John, M. 2011. *Reaction constraints for the pi-calculus : a language for the stochastic and spatial modeling of cell-biological processes*. Berlin: Logos Verlag.

Johnson, D. 2002. "A theoretician's guide to the experimental analysis of algorithms". In *Fifth and Sixth DIMACS Implentation Challenges*, edited by M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, 215–250. Providence: American Mathematical Society.

Kuhl, F., R. Weatherly, and J. Dahmann. 2000. *Creating computer simulation systems - An introduction to the High Level Architecture*. Prentice Hall, New York.

L'Ecuyer, P., and E. Buist. 2005, December. "Simulation in java with SSJ". In *Proceedings of the 2005 Winter Simulation Conference*, edited by M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 611–620. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

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*, edited by G. L. Steele, Jr., R. P. Gabriel, and R. Crocker, 345–352: ACM Press.

Merali, Z. 2010, October. "... why scientific programming does not compute". *Nature* 467:775–777.

Minar, N., R. Burkhart, C. Langton, and M. Askenazi. 1996, June. "The SWARM simulation system: a toolkit for building multi-agent simulations". Technical report, Santa Fe Institute.

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

Perumalla, K. 2005, June. "μsik: A Micro-kernel for Parallel/Distributed Simulation Systems". In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, edited by C. D. Carothers, S. J. Turner, and D. M. Nicol, 59–68. Washington, DC: IEEE Computer Society.

Praehofer, H., J. Sametinger, and A. Stritzinger. 2001. "Concepts and architecture of a simulation framework based on the JavaBeans component model". *Future Gener. Comput. Syst.* 17 (5): 539–559.

Röhl, M., and A. Uhrmacher. 2006, December. "Composing Simulations from XML-Specified Model Components". In *Proceedings of the 2006 Winter Simulation Conference*, edited by L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 1083–1090. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Rybacki, S., J. Himmelspach, E. Seib, and A. M. Uhrmacher. 2010, December. "Using workflows in M&S software". In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain,

J. Montoya-Torres, J. Hugan, and E. Yücesan, 535–545. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Rybacki, S., J. Himmelspach, and A. M. Uhrmacher. 2009, September. "Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata". In *First International Conference on Advances in System Simulation*, edited by W. E. Biles, K. Hawick, and L. Mönch, 62–67. Piscataway, NJ: The Institute of Electrical and Electronics Engineers, Inc.

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

Szabo, C., and Y. M. Teo. 2011, December. "An approach to semantic-based model discovery and selection". In *Proceedings of the 2011 Winter Simulation Conference*, edited by S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu, 3054 –3066. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Szyperski, C. 2002. *Component Software: Beyond Object-Oriented Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Uhrmacher, A. M., R. Ewald, J. Himmelspach, M. Jeschke, M. John, S. Leye, C. Maus, and M. Röhl. 2008, June. "One Modelling Formalism & Simulator is not enough! – A Perspective for Computational Biology Based on JAMES II". In *Proceedings of the the 1st FMSB Workshop*, edited by J. Fisher, Number 5054 in LNBI, 123–138. Cambridge, UK: Springer.

Valentin, E. C., A. Verbraeck, and H. G. Sol. 2003, June. "Effect of Simulation Building Blocks on Simulation Model Development". In *Proceedings of International Conference of Technology, Policy and Innovation*, edited by A.Ibarra, 54–61. Mexico.

Wilson, G. 2006. "Where's the real bottleneck in scientifc comuting". *American Scientist* 94 (1): 5.

Wilson, G. 2012. "Software Carpentry". Accessed June 1, 2012. http://www.software-carpentry.org.

AUTHOR BIOGRAPHY

JAN HIMMELSPACH is a post doc in the Computer Science Department at the University of Rostock. He received a diploma in Computer Science from the University of Koblenz and his doctorate in Computer Science from the University of Rostock. His research interests are in software engineering for modeling and simulation, credibility of modeling and simulation, and on efficient modeling and simulation solutions. He is the lead developer of JAMES II since 2003 and has supported the creation of more than 30 theses and of a number of projects with external partners based on the software since then. Before joining the University of Rostock he worked for LMD Innovative, a software company specialized on providing reusable components for software developers. His email address is jan.himmelspach@uni-rostock.de.