Proceedings of the 2008 Winter Simulation Conference
S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler eds.

A FLEXIBLE AND SCALABLE EXPERIMENTATION LAYER

Jan Himmelspach
Roland Ewald
Adelinde M. Uhrmacher

Albert-Einstein-Str. 21
University of Rostock
Rostock, 18059, Germany

ABSTRACT

Modeling and simulation frameworks for use in different application domains, throughout the complete development process, and in different hardware environments need to be highly scalable. For achieving an efficient execution, different simulation algorithms and data structures must be provided to compute a concrete model on a concrete platform efficiently. The support of parallel simulation techniques becomes increasingly important in this context, which is due to the growing availability of multi-core processors and network-based computers. This leads to more complex simulation systems that are harder to configure correctly. We present an experimentation layer for the modeling and simulation framework JAMES II. It greatly facilitates the configuration and usage of the system for a user and supports distributed optimization, on-demand observation, and various distributed and non-distributed scenarios.

1 INTRODUCTION

An experiment is the process of extracting data from a system by its inputs. This might include a series of trials with varying inputs. A simulation is an experiment with a model, or more precisely an execution of a model with concrete parameters. Hence, an experiment with a model usually consists of several simulation runs. Setting up an experiment typically comprises a multitude of steps, e.g., model selection, initialization, defining the observers, selecting the simulation engine, storing results, defining constraints for repetition, to name only a few. The more flexibility a simulation environment provides, the larger the degree of freedom the user has in setting up an experiment.

For reliable simulation results, it is also mandatory that experiments are repeatable. Thus, a crucial task of an experimentation layer is to ensure that all information for repeating an experiment is stored. As the degree of freedom offered by the system increases, fulfilling this requirement gets ever more challenging. With scalability we refer to this degree of freedom and the possibilities supported by the simulation framework to readily extend it. Thereby, we deviate from the usual notion of scalability in simulation as referring to one particular dimension, i.e., the ability of a simulation system to gracefully handle growing models of one type (which often coincides with limiting parallel processing overhead for large-scale applications, see Nicol et al. 2003).

In this sense, experimental layers of modeling and simulation environments that have been designed to conduct single experiments (Minar et al. 1996) or to evaluate individual simulation algorithms (e.g., Perumalla 2005) offer little reusability and scalability, although their simulation engine might do so. Being bound to single execution platforms (e.g., single machines, cluster, grids) or being restricted to single formalisms/languages, simulation algorithms, and data structures limits the scalability of the experimentation layer. Thus, a scalable simulation engine, as proposed by (Nicol 1998), is a key pre-requisite but not sufficient for realizing the scalable experimentation layer we have in mind.

What is the benefit of such an experimentation layer? A multitude of experiments is typically executed during the life-time of a model. The first experiments serve to explore the model, e.g., to optimize parameters, followed by experiments aimed at its validation, and finally the "real" experiments are conducted (Balci 2003). So the purposes – and thus, the types of experiments – even vary if we only have one model and one simulation engine. However, different simulation algorithms are required to ensure that the results to be interpreted are not biased by simulation algorithm artifacts (Edmonds and Hales 2003). Different application areas ask for different modeling formalisms, and different questions about the same system might only be answered by different models in different formalisms.

As already mentioned, the scalability of the experimental layer depends on the scalability of the simulation layer. The availability of parallel computing machines is constantly increasing, in the large (e.g., grid-based approaches) as well

978-1-4244-2708-6/08/$25.00

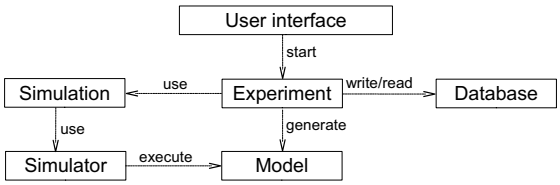

Figure 1: Framework controlled experiment execution

as in the small (e.g., multi-core processors), and with it the interest in this type of scalable simulation engines (Fujimoto 2007). With JAMES II, our first concern has also been to develop such a scalable simulation engine (Himmelspach and Uhrmacher 2004). By realizing a plug'n simulate concept, the user has gained further degrees of freedom in executing experiments and in evaluating new methodological concepts (Himmelspach and Uhrmacher 2007b). This concept is also used in developing the experimentation layer, so that scalability remains manageable for the user.

2 BACKGROUND

Different applications, created by different users in different modeling formalisms and languages, require a flexible environment – or the existence of a number of usable and maintained alternatives. Applications may range from small ones (e.g., a typical Lotka-Voltera model) up to large scale models (e.g., climatic changes in the world).

The purpose of a modeling and simulation environment is to conduct experiments with models. The term "experiment'" is therefore a notion of central importance. An "experiment" is defined by Cellier as "the process of extracting data from a system by excerpting it through its inputs." (Cellier 1991, page 4). Further on, we understand simulation as "an experiment performed on a model" (Korn and Wait (1978) after Cellier 1991, page 6). According to these definitions, an experiment is the central concept in such an environment.

2.1 JAMES II

The simulation framework JAMES II is a very lean system consisting of a set of core classes. The core of the simulation framework JAMES II is the central and most rarely changed part of the framework. The base for a scalable modeling and simulation framework is laid in the core. The main parts are: User interface, Data, Model, Simulator, Simulation, Experiment, and Registry. According to the previously given definition of an experiment, the `Experiment` package is central in the design of JAMES II (see Figure 1).

We used common software engineering techniques for the creation of the framework, e.g., the model-view-controller paradigm for decoupling its parts (Gamma et al. 1994). Another important design decision was to split model and simulation code completely. Thus, a simulator can access the interface of a model class but a model class is never allowed to access something in a simulator class. This makes it possible to switch the simulation engine (even during runtime) and to exchange the data structures used for the executable models – an essential feature for a scalable framework. In addition, this adds the possibility to use JAMES II for reliable evaluations of new simulation algorithms. In combination with an XML-based model component plug-in, this flexibility enables the freedom of choice in regards to model data type, simulator code (algorithm as such; or sub-algorithms, e.g., event queues), visualization, and runtime environment. The architecture is sketched in Figure 2. The layers depict the distance of a user from the packages.

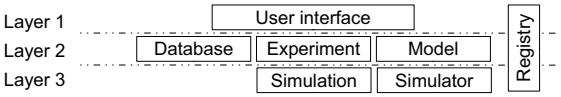

Figure 2: Packages of the simulation framework JAMES II

Extension points The "Plug'n simulate" approach (Himmelspach and Uhrmacher 2007b) has been developed for supporting, on the one hand, a variety of solutions which may be provided by third parties and, on the other hand, for enabling types of plug-ins yet unforeseen. Functionality not included in the core classes, especially modeling formalisms and simulation algorithms, can be extended by using plug-ins. The scalability of JAMES II relies on these extension points as well as on the availability of extensions for these. Extensions points in the core are, e.g., different modeling formalisms & languages.

2.2 Different users and interests

Different user groups may have completely different motivations for using modeling and simulation, e.g., teaching, exploration, validation, testing, optimization, or the evaluation of new algorithms. In addition they may have a different background – e.g., in regards to mathematics and modeling and simulation methodology. The usability of a system for different users is directly coupled to a barrier-free user interface and the support of an "intuitive" modeling formalism or language (from a specific user perspective). Even though this is not new, many simulation environments still only ship with one fixed user interface. JAMES II only provides a small user interface for basic tasks, and users are not condemned to use it. It may be completely replaced by another one (which allows, e.g., to integrate JAMES II seamlessly into other applications), and new user interfaces (or parts thereof) may be built upon the existing interface framework. Therefor the basic user interface employs the plug'n simulate scheme of JAMES II as well – e.g., different model editors can be integrated as plug-ins.

2.3 Different modeling formalisms & languages

The plethora of modeling formalisms nowadays imposes new challenges for developers of modeling and simulation environments. Most often developers have to decide which modeling formalism they are going to support, cutting off the majority of potential users. This is due to the circumstance that different modeling formalisms have different features, which eases the modeling of specific systems. Plug-ins for several modeling formalisms are available for JAMES II, among them Beta-binders, cellular automata, PDEVS, PdynDEVS, PepiDEVS, PdynEpiDEVS, SpacePi, and StochasticPi (Himmelspach and Uhrmacher 2007b). In addition, any model source can be used to feed JAMES II – as long as there is an appropriate reader being able to transform the model source into instances defined by a JAMES II model plug-in. Thereby, JAMES II supports implemented models as well as models stored in an arbitrary model source (e.g., XML files, databases). The support of a variety of modeling formalisms is not the only need for a flexible and scalable system: there is a need to be able to compose models to create more complex ones. Model composition is supported by a co-project of JAMES II: COMO (Röhl and Uhrmacher 2006). The integration of COMO is based on the model reader plug-in type as well. Modeling is a user- and system-specific task – thus, model editors should be exchangeable. JAMES II makes no restrictions in regards to the model editors which can be integrated. Model writer plug-ins for the used formalism allow to write the model to any supported target. Additionally, the usage of any external editor is possible as well – as long as there is a model reader able to read the output of the editor used.

2.4 A flexible and scalable simulation layer

An experiment in JAMES II defines the simulation runs to be executed with a single model, usually to answer a single question. A simulation layer which shall support a flexible and scalable experimentation layer has to support a variety of different simulation strategies. This comprises various algorithms and data structures for sequential and parallel simulation, as well as partitioning and load balancing support for the latter.

We have realized several plug-ins for these aspects, which can be replaced by each other. The algorithms can be selected automatically by the system.

Alternative algorithms In (Himmelspach and Uhrmacher 2006, Himmelspach et al. 2007), we introduced several algorithms to process PDEVS (Zeigler et al. 2000) models. They have been realized as plug-ins for JAMES II. These algorithms are: a direct implementation of the abstract simulator tree, a parallel sequential simulator (Himmelspach et al. 2007), and finally three sequential simulators – abstract sequential, flat sequential, and hierarchical sequential (Himmelspach and Uhrmacher 2006). The number of simulation algorithms per available modeling formalism can be extended by using the plug-in mechanism – thus the simulation layer of JAMES II can be easily adapted to special needs (e.g., arising from a hardware infrastructure or from an experiment definition).

Alternative data structures Maybe the most central data structure for discrete event simulations is the "event queue" (Vaucher and Duval 1975). Although this is well known in principle, many existing simulation environments ship with only one event queue implementation. More than ten different event queues have been realized for JAMES II, among them the calendar queue (Brown 1988) and the MList (Goh and Thng 2003). In (Himmelspach and Uhrmacher 2007a) we introduced several extended event queues for supporting an efficient simulation of PDEVS models. Our experiments underline the well known fact that there is no event queue which is best for the usage in all simulation algorithms or for all models – thus, for achieving real scalability of the simulation, and consequently of the experimentation layer, a set of event queues must be available in the system.

Alternative partitioning algorithms Setting up a parallel and distributed simulation of a model requires a partitioning algorithm for creating an initial partition of the given model. Due to differences in models, especially across different model descriptions, support for several partitioning algorithms is advisable. For example, a special algorithm for partitioning tree-based models (which may even contain constraints on which host a model part has to be placed) can be applied to PDEVS models (Ewald, Himmelspach, and Uhrmacher 2006), while other kinds of models are partitioned more efficiently by multi-level partitioning schemes as implemented in the METIS package (Karypis and Kumar 1995).

3 EXPERIMENTS

As can be seen in Figures 1, and 2, the user interacts with the simulation layer of JAMES II solely via defining and issuing experiments. Although the user interface enables creating and editing models as well, the actual execution shall be completely determined by the experiment description itself. This distinction allows to re-use models for different experimentation purposes, such as optimization or simulation studies. It also facilitates the management of experiment setups in general and increases the execution fidelity, as changes in the model do not lead to any changes in the experimentation procedure. We will now introduce the structure of experiment definitions and detail their processing.

3.1 Experimental Structure

In JAMES II, experiments are instances of the `BaseExperiment` class. First of all, an experiment is defined with respect to a model, so it needs to store the appropriate factory class for model reading and optional parameters (such as, e.g., a flag to enable some basic checks on the model's soundness while reading). Model readers are a plug-in type in JAMES II, so that all model reader factories share the same interface. The model itself is *not* a part of the experiment definition, as it could be very large and the experiment might need to be transferred to a remote machine. Furthermore, the user can define input variables to be modified in the course of the experiment. By doing so, many different simulation settings can be defined conveniently. The variables and their modification rules are also part of the experiment definition.

Being JavaBeans, instances of *BaseExperiment* can be easily serialized and, e.g., stored to the file system or a database. Reading and writing experiments is yet another plug-in type, so new ways to load and store experiments may be implemented at will. Currently, JAMES II supports two different XML formats. For more advanced studies, one could easily create a subclass of `BaseExperiment` and still use the rest of the system, it is therefore extensible to more specific purposes. Experiments can be compiled to so-called *experiment suites*. This is an easy way to define sets of related experiments, e.g., experiments to be done with a certain model. While all of these features are straightforward, there are several more complex issues that need to be supported by a scalable experimentation environment:

Data Storages JAMES II provides a plug-in type for components that allow to store simulation data. As a user may require anything from local files to (distributed) data base storages, the selected factory and its parameters are part of the experiment definition. This provides scalability from a storage perspective. Data storages store simulation data in a generic format and assign a unique ID to every experiment and simulation run. Until now, we realized data storages for local memory, files, and databases accessible via JDBC (**?**). How the simulation data is obtained is described in Section 3.2.

Random Numbers The generation of pseudo random numbers is a commonplace requirement in discrete-event modeling, since many models incorporate stochastic features. Unfortunately, the random number generators (RNGs) provided by standard software libraries may not always meet the requirements in large-scale stochastic applications (Matsumoto et al. 2007). Hence, a plug-in type for RNGs was implemented, which allows to configure the type of the RNG algorithm (and, if necessary, its parameters) in the experiment definition. This makes it possible to execute the model at hand with different RNG algorithms, so that resulting artifacts can be detected (Ewald et al. 2008).

Replication Criteria When working with stochastic models, users need to conduct numerous simulation replications to get valid insights into the model's behavior. The number of replications that is sufficient to get, e.g., results with a certain statistical significance, may vary. This implies that the number of required replications is often not just a constant, but shall be calculated dynamically, based on the model's behavior in previous runs (Law and Kelton 1999). To allow this kind of automatic experiment control, we implemented a plug-in type for different realizations of *replication criteria*. Each experiment can be associated with an arbitrary number of those, and an additional replication will be issued if *any* of the associated criteria demands it.

Simulation Runners Finally, the user also needs to decide in which way the experiments shall be conducted. As JAMES II supports various kinds of fine-grained distributed simulation (Himmelspach et al. 2007), the experimentation layer also supports sequential, fine-grained, and coarse-grained distributed simulation. A fine-grained distributed execution distributes a *single* simulation run over a set of processors, whereas a coarse-grained approach distributes complete simulation runs, which are then executed sequentially on the remote machine. A *simulation runner* is a mechanism that realizes any of those execution strategies. It is the co-operation between these two complimentary objects, experiment and simulation runner, which is one of the main features characterizing our approach: it encapsulates all of the execution logic within a simulation runner, while all user options and configuration elements reside in the experiment. This distinction allows to interface an experiment with simulation runners optimized for specific hardware platforms (e.g., clusters, multi-core CPUs) or configured to use existing infrastructures (e.g., existing Grid toolkits like Globus). Again, not the simulation runner itself but a factory to create it (and optional parameters) are stored within the experiment.

Repeatability A very important aspect of experiments is repeatability. Usually, results of an experiment are only regarded trustworthy if an experiment with identical inputs leads to identical results. This is not always easy to achieve, because the results of an experiment may depend on stochastic effects, the hardware, the programming language in which model or simulator are implemented, the simulation algorithm as such, and so on.

Important for repeatability is the validation of simulation algorithms. If simulation algorithms are carefully validated, the usage of different simulation algorithms should not be a problem (for discrete event simulations). For differential equation solvers this is different: here, the algorithm as such plays an important role (and additionally, the precision of data types).

The problem of repeatable random numbers, required to realize stochastic elements in the model, can be solved by making seeds and parameters of random number generators configurable. This is a prerequisite for debugging stochastic simulations and ensuring their repeatability. However, there are often many probability distributions to be sampled in a distributed simulation, which implies the use of many RNG instances in different threads. This problem is known as parallel random number generation (Srinivasan et al. 2003, Mascagni and Srinivasan 2004), and requires careful RNG initialization to avoid correlations. We defined an RNG generator plug-in type, so that parallel RNGs can be created by various methods (e.g., parameterization, cf. Mascagni and Srinivasan 2004) and these are separated from the rest of the system. Parameters for RNG generators can be set in the experiment definition.

Thus, an experiment must carry more than only the information about inputs to be used to stress a model. The experiment definition of JAMES II is currently restricted to inputs, random parameters and (if wanted) to the fixed selection of algorithms. Additional information about a run (e.g., the versions of the used algorithms) can be collected automatically and stored together with the simulation results.

3.2 Configuring Observations

While observation of simulation data is a time-consuming task, the degree of observation largely depends on the model aspects the user is interested in. Thus, there is an essential need for a scalable observation mechanism.

JAMES II achieves this scalability on several levels: Firstly, we discriminate between observation and instrumentation. An instrumenter instantiates observers for all relevant parts of the model. It may select specific model properties to be observed or even instrument everything (which is the worst case from a performance perspective). As instrumenters are configurable by the user, this scales well with respect to the amount of simulation data that is actually needed. Secondly, the design of the observer mechanism allows the usage of "mediators", which gather the collected information before they are sent across a network. This allows to reduce the network load and also decouples model execution from data transmission. Thirdly, the data collection can be done with any of the aforementioned data storages. Instrumenters for both models and simulators are also part of the experiment description, and optional parameters (such as the type of all sub-models to be observed) can be given.

4 EXECUTING EXPERIMENTS

From a programming perspective, an experiment can be regarded as a means to generate the set of parameter configurations to be executed. As described in Section 3, experiments also define further requirements (like the destination to which observation data shall be written), but this additional information is simply needed to define the way in which one configuration shall be executed. We call the combination of model parameters and additional settings a *simulation configuration* in the following. A simulation configuration is therefore a complete set of information needed to simulate a model with a single set of parameters. Most experiments require to execute multiple simulation configurations, e.g., to test the sensitivity of different model parameters.

The capability of experiments to generate simulation configurations needs to be complemented with a component that processes these configurations. This approach resembles a so-called client/server architecture, in which an experiment acts as the client and generates requests (simulation configurations), while the server receives the requests and processes them. The advantages of this approach in our setting are twofold: At first, it decouples the execution logic from experiment definition and therefore makes the whole process of experiment execution transparent to the client. By doing so, we achieve a considerable amount of scalability with respect to execution modes and ways of generating configurations (see Section 4.1). Secondly, it allows to execute several experiments in parallel, as a server is able to handle more than one client. Two additional problems arise when considering the various use cases of an experimentation layer.

Iterated Processing There are several scenarios, e.g., when integrating optimization algorithms to steer experimentation, that require to schedule new simulation configurations after considering the results of *past* runs. This implies that an experiment might not be able to generate all simulation configurations at once, but only a certain portion that is needed to come up with further setups to be evaluated. Therefore, the server, which is the simulation runner as described in Section 3.1, must not only receive and process simulation configurations, but also needs to notify the experiment upon completion. Then, the experiment might access the corresponding data storage, analyze the output, and schedule new tasks. This continuous interaction between experiment and simulation runner allows to integrate any kind of automatic parameter space exploration techniques. The experimentation proceeds until the experiment generates no additional configurations.

Separate Execution Control Sometimes, users might need to interact with the experimentation layer, e.g., when deciding which kind of simulation output shall be used for visualization at runtime. It might also be necessary to direct the algorithms that generate new configurations (e.g., as described by Persson, Grimm, and Ng 2006), and to stop experiments or simulation runs manually. Then again, there are many scenarios where such interaction is not desired (e.g., for large-scale batch experiments) and there might

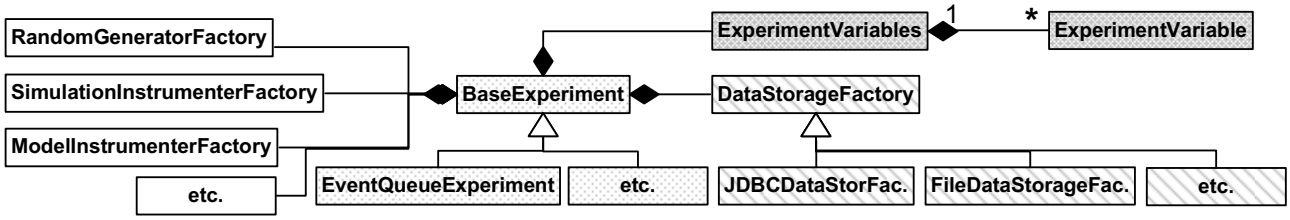

Figure 3: Central classes in the experimentation layer: `BaseExperiment` and its descendants (dotted background) form the central entities. They control data structures to generate new model inputs (`ExperimentVariables`, dark grey) and can be configured with any descendant of a required factory, e.g., any `DataStorageFactory` (diagonal lines).

even be no graphical user interface at all. To make our approach scale with the amount of user interaction that is desired, we added an *execution controller* to the client/server setup. The final setup and interaction protocol is outlined in Figure 4.

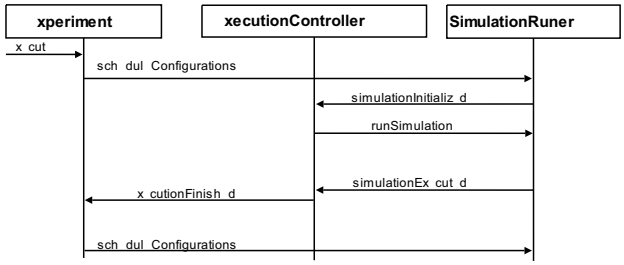

Figure 4: Interacting threads in the experimentation layer: experiment, execution control, and simulation runner.

When the experiment gets executed, it uses the predefined factory to create a simulation runner in a separate thread. It then proceeds by generating as many simulation configurations as possible, and sends the whole list to the simulation runner. The simulation runner is now responsible for repeatedly executing each simulation configuration as defined by the replication criteria. The order of initialization and the mode of execution (i.e., parallel v. sequential) are *not* predetermined. For each execution of a simulation, the runner only has to do three steps in a specified order (see Figure 4): Firstly, it shall *notify* the execution controller when a simulation is set up and can be started. It does so by sending a *runtime information* object containing references to all the observers it instantiated, and which the user might now use for on-line visualization. Secondly, the runner needs to wait until it gets the command from the execution controller to proceed with executing the simulation run. This is necessary to enable the selection of observers for on-line visualization. If the simulation runner would not wait for the execution controller, then the user could miss important output at the beginning of the simulation. In a non-interactive mode, the control thread may automatically notify the runner to proceed. Finally, the runner needs to execute the simulation run and notify the execution controller, which in turn notifies the experiment. If all simulation configurations are completed, the experiment may generate new configurations and send them to the simulation runner.

This setup allows to scale the experimentation layer in any required direction: as all components are solely interacting over their interfaces, each participating element might be replaced by another object implementing the same interface. In fact, we have implemented two variants of the simulation runner, two versions of the execution controller (one to work in command-line mode, one to work in our default GUI), and even several subclasses of our basic experiment class.

4.1 Usage scenarios

The scalable experiment layer can be used for a broad set of different experiments, among them: small scale (sequential) as well as large scale (many runs or large models) simulation runs (coarse/fine grained parallel), evaluation of algorithms, teaching, validation, and optimization. Examples for the evaluation of algorithms can be found in (Himmelspach and Uhrmacher 2006, Himmelspach and Uhrmacher 2007a). Two simple examples for simulation studies are described in the next section.

4.2 Examples

Model analysis Consider a simple queuing system consisting of a generator, a buffer, and a processor, all realized as PDEVS models. The generator generates new jobs with a given rate. These are then stored in the buffer, where they reside until they get processed by the processor (at another given rate). Now, it may be of interest to analyze the mean buffer size, given the (possibly stochastic) rates for generating and processing jobs. In JAMES II, several probability distributions are available (e.g. exponential, uniform, biased, triangular, etc. (Himmelspach 2007)), which can be easily used in this context.

This experiment definition does not preselect a simulator or any further parameters (e.g., event queues to be used). The decision of which to use is left over to the simulation framework. The parameters are defined as "loops", i.e., for each generator rate every available processor rate is

Model to be used: `examples.pdevs.genbufproc`
Parameters:
- GeneratorRate: next from *DistributionSequence* (initialized with next random seed)
  - ProcessorRate: next from *DistributionSequence* (initialized with next random seed)

Simulation end time: 10000
Replications: 10 (repeat each experiment 10 times)
Data sink: `datastorage.jdbc.JdbcDataStorageFactory`
Model instrumenter: `examples.pdevs.genbufproc.instrumenter.GenBufProcInstrumenter`
Simulation runner: Sequential simulation runner

Figure 5: Definition of the first example. An excerpt of its corresponding XML-description can be found in Figure 6.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_02" class="java.beans.XMLDecoder
    ">
<object class="james.core.experiments.BaseExperiment">
<void property="dataStorageFactory">
 <object class="datastorage.jdbc.
     JdbcDataStorageFactory"/>
</void>
<void property="experimentVariables">[...]</void>
<void property="modelInstrumenterFactory">
 <object class="examples.pdevs.genbufproc.
     GenBufProcModelInstrumenterFactory"/>
</void>
<void property="modelLocation">
 <object class="java.net.URI">
 <string>java://examples.pdevs.genbufproc.GenBufProc</
     string>
</object>
</void>
<void property="parameters">
 <void property="endTime"><double>10000.0</double></
     void>
</void>
<void property="replicationCriteria">
 <void method="add">
 <object class="james.core.experiments.replication.
     ReplicationNumberCriterion">
 <void property="numOfReplications"><int>10</int></
     void>
 </object>
</void>
</void>
</object>
</java>
```

Figure 6: Excerpt of an XML-based experiment definition. Besides such a bean-serialization, it is possible to implement additional experiment reader/writer plug-ins, e.g. to access experiment databases.

used. Each of these simulation runs shall be repeated 10 times. Even though this model is pretty small, scalability can be very essential for a simulation study. There are eight different probability distributions available, i.e., we have to execute $8 \times 8\times = 64$ different setups, each of them tenfold. In the given configuration these are already 640 simulation runs. If we now decide to execute more replications, e.g., 100, we'd end up with 6400 runs. Adding

Model to be used: `examples.pdevs.genbufproc`
Parameters:
- Simulator: flat sequential
- EventQueue: next from *list of registered event queues*
  - GeneratorRate: next from *DistributionSequence* (init with next rand seed)
    - ProcessorRate: next from *DistributionSequence* (init with next rand seed)

Simulation end time: 10000
Replications: 10 (repeat each experiment 10 times)
Data sink: none; Model instrumenter: none
Simulation runner: Sequential simulation runner

Model to be used: `examples.pdevs.forestfire`
Parameters:
- Simulator: flat sequential
- EventQueue: next from *list of registered event queues*
  - Width: increment by 10 from 10 to 200
    - Height: increment by 10 from 10 to 200

Simulation end time: 10000
Replications: 10 (repeat each experiment 10 times)
Data sink: none; Model instrumenter: none
Simulation runner: Sequential simulation runner

Figure 7: Experiment suite definition

additional distributions or parameterizing the existing ones can easily result in several thousand additional executions.

Although a small set of simulation configurations may be executable on a single computer quickly, a larger set should be simulated using several machines. To do so, merely one little change has to be applied to the experiment configuration: another (parallel) simulation runner needs to be chosen.

Algorithm analysis If algorithms shall be evaluated a large number of simulation runs may have to be executed. For evaluation experiments with algorithms the usage of an experiment suite makes sense: the evaluation should be done with a number of models having different characteristics whereby there might be diverse parameters to be explored per model.

Our suite will consist of two simple experiments, the first experiment uses the model from the example given above, the second experiment is based on a forest fire model already used in several publications (e.g. Himmelspach and Uhrmacher 2007a).

The first experiment defines $640 \times eventqueues$ experiments, the second experiment $20 \times 20 \times eventqueues$ experiments. By using the sequential simulation runner all these experiments will be executed on a single host. If a new event queue is added to the system the experiment suite can be re-executed, and the new event queue can be easily evaluated in direct comparison to the other event queues in the system. The experiment definition can be easily adapted to add the dimension of different simulation algorithms us-

ing the event queues – e.g. we could replace the fixed simulator entry by a generic list (containing all registered PDEVS simulators). The results of these experiments can be found in (Himmelspach and Uhrmacher 2007a).

5 SUMMARY

A scalable experimentation layer must be open for a large variety of different types of experiments and ways of conducting these. Thus, flexibility and scalability of a well defined and highly reusable experimentation layer are not restricted to a single dimension: such a layer has to allow the experimentation with models of arbitrary type and size in "any" language, on different hardware infrastructures and with different goals. Especially the differing goals enforce an integrated solution for an experiment layer: only if the experiment layer has full access to all experiment-related information (including results from previous simulation runs) any type of experiment can be efficiently supported. Additionally, there should not be any restrictions regarding potential visual support (setup and analysis), data collection, etc.. As any experiment definition, our experiment layer has to provide means to unambiguously specify and store configurations, so that repeatability of experiments can be ensured. With the proposed layout, we achieve implementation flexibility (and thus, potential scalability) for the following aspects of simulation execution:

- Various types of models (discrete, continuous, hybrid), various model languages, various models, various model sources
- Various kinds of algorithms (different kinds of execution, e.g. sequential or distributed)
- Configuration generation (different kinds of experiment definitions, from various sources)
- Degree of observation / flexible instrumentation
- Data storages (different kinds of output storages, e.g. databases)
- Decision making about further replications (Replication Criteria, different kinds of output/certainty)
- Parallelization / interfacing the Grid (different infrastructures)
- Off- and Online Visualization (different user interaction patterns)
- Various kinds of methods to define experiment parameter space implicitly (optimization, validation, etc.)

6 OUTLOOK

The simulation layer as realized for JAMES II does not only support a scalable simulation architecture but also the experimental evaluation of competing simulation algorithms. The advantage of this layer is that a new algorithm can be easily plugged into the system and can then be compared to other available algorithms. This makes algorithm performance comparisons more reliable, as no algorithm needs to be implemented twice. The experiments with alternative implementations also indicate that there is no algorithm that always delivers best performance. Thus, a really scalable framework must contain as many different solutions (e.g., algorithms) as possible. Extensive simulation studies require a flexible and well-defined experimental setup and a reliable and well tested simulation environment. This is especially true if models shall be experimentally validated. JAMES II provides the preconditions for this and thus future work will additionally deal with the integration of automatic model validation. In the near future, we will add a grid-inspired approach to JAMES II, i.e., massively parallelize the execution of simulations. Additionally, we are currently extending the scalability of the simulation data collection, because a fast execution of a simulation can easily be hampered by slow data collection.

7 ACKNOWLEDGMENTS

This research is supported by the DFG (German Research Foundation).

REFERENCES

Balci, O. 2003. Verification, validation, and certification of modeling and simulation applications: verification, validation, and certification of modeling and simulation applications. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, ed. S. E. Chick, P. J. Sanchez, D. M. Ferrin, and D. J. Morrice, 150–158: Winter Simulation Conference.

Brown, R. 1988. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Commun. ACM* 31 (10): 1220–1227.

Cellier, F. E. 1991. *Continuous system modeling*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

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

Ewald, R., J. Himmelspach, and A. M. Uhrmacher. 2006. Embedding a non-fragmenting partitioning algorithm for hierarchical models into the partitioning layer of James II. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, ed. L. F. Perrone, B. G. Lawson, J. Liu, and F. P. Wieland, 848–855: Winter Simulation Conference.

Ewald, R., J. Rössel, J. Himmelspach, and A. M. Uhrmacher. 2008. A plug-in - based architecture for random number generation in simulation systems. In *WSC '08: Proceedings of the 40th conference on Winter simu-*

lation, ed. S. J. Mason, R. R. Hill, L. Moench, and O. Rose: Winter Simulation Conference.

Fujimoto, R. M. 2007, March. Keynote: Modeling, simulation and parallel computation: The future is now. In *Proceedings of the 2007 Spring simulation multiconference*, Volume 1, vii–viii: SCS.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, USA.

Goh, R. S. M., and I. L.-J. Thng. 2003, Dezember. Mlist: An efficient pending event set structure for discrete event simulation. *International Journal of Simulation - Systems, Science & Technology* 4 (5-6): 66–77.

Himmelspach, J. 2007, December. *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations und Experimentiersystems - Entwicklung und Evaluation effizienter Simulationsalgorithmen*. Reihe Informatik. Göttingen: Sierke Verlag.

Himmelspach, J., R. Ewald, S. Leye, and A. M. Uhrmacher. 2007, March. Parallel and distributed simulation of Parallel DEVS models. In *Proceedings of the DEVS Integrated M&S Symposium*, 249–256.

Himmelspach, J., and A. M. Uhrmacher. 2004. A component-based simulation layer for JAMES. In *Proc. of the 18th Workshop on Parallel and Distributed Simulation (PADS), May 16-19, 2004, Kufstein, Austria*, 115–122.

Himmelspach, J., and A. M. Uhrmacher. 2006, Oct. Sequential processing of PDEVS models. In *Proceedings of the 3rd EMSS*, ed. A. G. Bruzzone, A. Guasch, M. A. Piera, and J. Rozenblit, 239–244. Barcelona, Spain.

Himmelspach, J., and A. M. Uhrmacher. 2007a, March. The event queue problem and PDEVS. In *Proceedings of the DEVS Integrated M&S Symposium*, 257–264: SCS.

Himmelspach, J., and A. M. Uhrmacher. 2007b, March. Plug'n simulate. In *Proceedings of the ANSS*, 137–143: IEEE Computer Society.

Karypis, G., and V. Kumar. 1995. *MeTis: Unstrctured graph partitioning and sparse matrix ordering system, version 2.0*.

Korn, G. A., and J. V. Wait. 1978. *Digital continous-system simulation*. Englewood Cliffs, N.J.: Prentice-Hall.

Law, A., and D. W. Kelton. 1999, December. *Simulation modeling and analysis (industrial engineering and management science series)*. McGraw-Hill Science/Engineering/Math.

Mascagni, M., and A. Srinivasan. 2004, July. Parameterizing parallel multiplicative lagged-fibonacci generators. *Parallel Computing* 30 (7): 899–916.

Matsumoto, M., I. Wada, A. Kuramoto, and H. Ashihara. 2007, September. Common defects in initialization of pseudorandom number generators. *ACM Trans. Model. Comput. Simul.* 17 (4): 15 (1–20).

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.

Nicol, D. M. 1998. Scalability, locality, partitioning and synchronization pdes. In *Proceedings of the twelfth workshop on Parallel and distributed simulation*, 5–11: IEEE Computer Society.

Nicol, D. M., J. Liu, M. Liljenstam, and G. Yan. 2003. Simulation of large scale networks I: simulation of large-scale networks using SSF. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, ed. S. E. Chick, P. J. Sanchez, D. M. Ferrin, and D. J. Morrice, 650–657: Winter Simulation Conference.

Persson, A., H. Grimm, and A. Ng. 2006. On-line instrumentation for simulation-based optimization. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, ed. L. F. Perrone, B. G. Lawson, J. Liu, and F. P. Wieland, 304–311: Winter Simulation Conference.

Perumalla, K. 2005, June. μsik: A micro-kernel for parallel/distributed simulation systems. In *ACM/IEEE/SCS Workshop on Parallel and Distributed Simulation (PADS)*. Monterey, CA: IEEE Computer Society Press.

Röhl, M., and A. M. Uhrmacher. 2006. Composing simulations from XML-specified model components. In *WSC '06: Proceedings of the 38th conference on Winter simulation*, ed. L. F. Perrone, B. G. Lawson, J. Liu, and F. P. Wieland, 1083–1090: Winter Simulation Conference.

Srinivasan, A., M. Mascagni, and D. Ceperley. 2003, January. Testing parallel random number generators. *Parallel Computing* 29 (1): 69–94.

Vaucher, J. G., and P. Duval. 1975. A comparison of simulation event list algorithms. *Commun. ACM* 18 (4): 223–230.

Zeigler, B., H. Praehofer, and T. Kim. 2000. *Theory of modeling and simulation*. London: Academic Press.

AUTHOR BIOGRAPHIES

JAN HIMMELSPACH is employed as a (post-doc) researcher in the Modeling and Simulation Group of Prof. Uhrmacher. His research interest is simulation software engineering and efficient algorithms for simulation.

ROLAND EWALD is a PhD candidate employed in the Modeling and Simulation Group of Prof. Uhrmacher. His research interest is the automatic configuration of simulation software.

ADELINDE M. UHRMACHER is head of the Modeling and Simulation Group at the University of Rostock. Her research interests are in modeling and simulation methodologies, and their applications.