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

EXPERIMENTAL ANALYSIS OF LOGICAL PROCESS SIMULATION ALGORITHMS IN JAMES II

Bing Wang
Yiping Yao

School of Computer Science
National University of Defense Technology
410073, Changsha, P.R. China

Jan Himmelspach
Roland Ewald
Adelinde M. Uhrmacher

Institute of Computer Science
Albert-Einstein-Str. 21
18059 Rostock, Germany

ABSTRACT

The notion of *logical processes* is a widely used modeling paradigm in parallel and distributed discrete-event simulation. Yet, the comparison among different simulation algorithms for LP models still remains difficult. Most simulation systems only provide a small subset of available algorithms, which are usually selected and tuned towards specific applications. Furthermore, many modeling and simulation frameworks blur the boundary between model logic and simulation algorithm, which hampers the extensibility and the comparability. Based on the general-purpose modeling and simulation framework JAMES II, which has already been used for experiments with algorithms several times, we present an environment for the experimental analysis of simulation algorithms for logical processes. It separates model from simulator concepts, is extensible (in regards to the benchmark models, the algorithms used, etc.), and facilitates a fair comparison of algorithms.

1 INTRODUCTION

Many algorithms have been proposed in modeling and simulation during the last decades, but most of them have merely been implemented in the modeling and simulation systems created around these algorithms (Perumalla 2005). From the view of the "Experimental Algorithmics" community (McGeoch 2001, McGeoch 2007, Johnson 2002), this makes a fair comparison of the algorithms difficult or even impossible. In addition, many simulation systems do not separate model and simulation algorithm instances clearly. This leads to a questionable mixture, which cannot guarantee that the simulation algorithm works correctly for all kinds of valid models. According to (Zeigler, Praehofer, and Kim 2000), the separation of model and simulator should allow the exchange of both parts independently from each other. One could think of alternative implementations of a model specification (all sharing a predefined interface), and the same holds for simulators: there might be a variety of simulators for each model interface. Providing different implementations of the same algorithm is also a prerequisite for *validating* simulators against each other, which reduces the risk of invalid simulation outcomes. A thorough and reliable evaluation of existing solutions is therefore highly demanded, at best in an open environment which allows others to reproduce the obtained results later on. This should also help to overcome the various pitfalls one encounters when analyzing algorithms empirically (e.g., cf. (Johnson 2002, Gent et al. 1997)).

The term *logical process* (LP) was introduced to parallel and distributed discrete-event simulation (PDES) as a metaphor for the physical processes a system consists of. It is widely used as a modeling paradigm in the domain of PDES. Sometimes it is referred to as a synonym for any sequential simulation or simulator (Fujimoto 2000, p. 40), and it has been introduced to sequential simulation branching (Curry et al. 2005, Peschlow, Geuer, and Martini 2008). In time-parallel simulation, which decomposes the model temporally, every computation unit is also called LP (Fujimoto 2000, p. 177 et sqq.). We regard LPs as the core units of execution during the simulation (Chandy and Misra 1979, Misra 1986). They interact with each other by exchanging *events*. This definition of LPs as model entities is common in the field of parallel and distributed discrete-event simulation, e.g., in (Fujimoto 2000).

We also would like to emphasize here that there are other levels of parallelism that can be exploited for parallel simulation, e.g., executing replications in parallel (e.g., (Leye et al. 2008)), or by conducting parallel computations *within* an LP. This paper focuses on parallelism on the level of LPs, but the environment can be used to evaluate the other variants as well - e.g., the coarse-grained execution is automatically supported for all simulation algorithms.

2 BACKGROUND AND MOTIVATION

Over the last decades, many solutions for the simulation of LP-based models have been proposed (Fujimoto 2000). They usually aim at overcoming the performance bottleneck of a sequential discrete-event simulation (Misra 1986), i.e., to achieve a *speed-up* in comparison to a sequential simulation. There are several associated techniques, for example, *partitioning* and *synchronization*, which are two important issues that strongly influence the performance of a parallel and distributed simulation: A partitioning policy has to decide how the LPs, i.e., the model entities, are mapped onto the available processors that shall execute them. This is a fundamental problem in parallel and distributed computing (Hendrickson and Kolda 2000).

Of even more importance is the *synchronization* protocol. Since LPs are executed concurrently in a PDES, communication among them has to be synchronized. The existing protocols can be characterized as conservative, optimistic, or hybrid (Jha and Bagrodia 1994, Perumalla 2005), and they have a strong impact on PDES performance. More details on synchronization are given in Section 5.

Many existing PDES solutions have been proposed for specific hardware architectures, e.g., (Jefferson et al. 1987, (Fujimoto and Hybinette 1997, Chen and Szymanski 2005, Perumalla 2007)). Tuning algorithms towards specific hardware hampers the reproducibility of obtained performance results on other platforms, but is often inevitable for delivering good performance for the application at hand. Today's hardware architectures with their long CPU pipelines and multiple cache layers exhibit non-trivial runtime behavior. This makes the performance analysis of algorithms executed on them a non-trivial task (McGeoch 2007). Consequently, computational complexity theory is often insufficient for assessing an algorithm's real-world performance (LaMarca and Ladner 1997). Still, the theoretical analysis of algorithms may provide valuable insights regarding certain design decisions, although it usually requires strong assumptions and the results are rather general or inconclusive (e.g., (Nicol 1998, Gupta, Akyldiz, and Fujimoto 1991)). The importance of empirical algorithm performance analysis has been stressed by the "Experimental Algorithmics" community over the last years, which deals with its methodological issues (McGeoch 2007, Johnson 2002). Following their guidelines regarding a fair comparison of algorithms, it is essential to compare all algorithms based on the *same* benchmark models, and to only compare algorithms that are executed on the *same* platform. Moreover, a single implementation should not be trusted too much – it might turn out to be erroneous if thoroughly validated. In addition, it could rely on some operation that acts as a performance bottleneck – which could be avoided, and which thus renders the results achieved not to be valid in general. This is particularly true for parallel simulation algorithms, as they are challenging to implement and inconsistencies can be very hard to find. To make things worse, most research papers present new algorithms compared to only one or two old ones, and the researchers might simply have spent more time on optimizing their own algorithm. All these inaccuracies are hard to avoid and may shed doubt on the obtained results.

Hence, it is neither sufficient to only compare two algorithms in theory, nor to compare their performance across different platforms, programming languages, and simulation systems – which is a central motivation for an experimental environment that ensures a fair comparison. Recent emergence of new application demands, techniques, and hardware platforms results in enhancements to traditional techniques and the formulation of new methods for parallel and distributed simulation (Perumalla 2006). Thus, research on PDES algorithms could benefit from a flexible and efficient way to evaluate and compare old and new PDES algorithms.

JAMES II provides a fixed experimental setup and the extensibility for new algorithms by design, and thus forms a solid base for the experimental analysis of simulation algorithms (Himmelspach and Uhrmacher 2007b). Anyone can add an implementation of a certain algorithm to the system and compare it to the performance of existing alternatives. Thus developers can concentrate on "their" algorithms, and while time passes the list of available, and hopefully tuned, implementations increases constantly. In addition anyone can add new benchmark models: the performance of a certain algorithm may strongly depend on the concrete input (here: a model) it is applied to. The more and better benchmark models there are, the more reliable are the experimental results. Still, in the first instance the most commonly used models should already be provided – only this makes it possible to compare newly obtained results from our framework to those that are already documented in the literature. Detailed descriptions of the experimental setup form the base for a sound experimental analysis, which is provided by the JAMES II experimentation layer (Himmelspach, Ewald, and Uhrmacher 2008). These details make it easier to discuss and compare results, and may make them "future safe", i.e., a reliable source of information for future research. Especially the latter aspect requires the usage of abstract, normalized performance measurement values instead of providing just some general information about the platforms used (Johnson 2002). We believe that work on re-evaluation and standardized benchmarking may help to ensure the research quality of the PDES community, an important issue that has recently been recognized as crucial for the M&S community in general (Smith et al. 2008).

3 RELATED WORK

There have already been several attempts to provide a solid base for the comparative analysis of simulation algorithms following the logical process paradigm, albeit for certain specific algorithm families, such as synchronization algorithms. One example for this is WARPED (Martin et al. 2003), which was intended to be *"freely available to the research community for analysis of the Time Warp design space."* (Martin, McBrayer, and Wilsey 1996, p. 1). Other simulation systems, such as APOSTLE (Wonnacott and Bruce 1996), focused on the impact of granularity, i.e., the time ratio between executing a single event and transferring it to another processor. In (Ferscha, Johnson, and Turner 2001), a thorough performance comparison for several well-known synchronization algorithms is conducted. The selected algorithm variants are carefully realized by activating certain code segments. While it reduces the code redundancy for the realized variants, it does not provide a flexible plug-in architecture for testing new methods, i.e., it is not easily extensible. Several theoretical performance analyses have been conducted for PDES as well, e.g., (Nicol 1998, Gupta, Akyldiz, and Fujimoto 1991). While they provide important insights into the nature of PDES, their assumptions are often quite strong, which prevents their results from being easily transferable to real-world systems. As our environment is built on top of JAMES II, it is implemented in Java. Java is nowadays a pretty common choice for the implementation of modeling and simulation tools. Its advantages are its ease of use, platform independence, as well as its built-in support for multi-threading and serialization. Consequently, there are several PDES systems programmed in Java, for example JTED (Cowie 1998), JWARP (?? 1998), SPADES/JAVA (Teo and Ng 2002), IDES (Nicol et al. 1998), CLUSTERSIM (Goes, Ramos, and Martins 2004), and even web-based systems (Ferscha and Richter 1997). This (incomplete) list should illustrate that Java is a common and accepted language in PDES research. Some concerns may arise because Java bytecode is interpreted by a Java virtual machine (JVM), which means that it cannot directly exploit special hardware features and also degrades the performance of Java programs. On the other hand, specific hardware could be exploited on the level of the JVM or the libraries that are used, so that this dependency can also be regarded as another impact factor to be controlled, very much like the operating system underneath. Concerns regarding runtime performance are partially true, but we think that ease of implementation (e.g., due to garbage collection), free access, and the fact that a comparison is still fair, as long as *all* algorithms are implemented in Java, are good arguments to use it as a base for such an environment. Even more so as the performance gap to C/C++ can be narrowed by using just-in-time compilers etc. Finally, Java is freely available and widespread – essential prerequisites for an experimentation environment that aims at integrating research efforts from others.

4 MODELING IN THE LOGICAL PROCESS PARADIGM

A model expressed in the logical process paradigm solely consists of "logical processes" as atomic model entities, and their interconnections. Hence, a logical process model can be represented as a graph $M = (LP,C)$, with LP being the set of nodes, each of which represents a logical process, and C being the set of edges to denote communication between the LPs.

JAMES II provides support for different ways of getting an instantiated version of the model to be simulated. The first and recommended one is to use an explicit modeling language. Then, getting an executable model of a system can be split up into two steps: the first one is the creation of a symbolic model (in the specialized modeling language), the second one is the conversion of the created model into an executable one. This process has two major advantages: (1) The symbolic model can be independent of platform, programming language, and simulation system. (2) The executable model can be based on different implementations of the data structures used internally. Thereby executable models can either be dynamically generated in memory (by using predefined classes, in combination with "interpreting" state change rules), or by generating new source code for the models, compiling the source, and executing the result. This transformation is done in JAMES II by using a "model reader" mechanism. The second possibility is to retain to coding models directly in Java (based on the executable model interfaces). However, here you loose the possibility to exchange the model's data structures easily: you have to recode the model. Due to the lack of a specialized language for LP based models so far, we have to use this mechanism for now. But this is no serious problem, because these models can coexist with others defined in any language for LP-based models later on. A well-defined interface of models provides the flexibility to evaluate different combinations of model and simulator implementations. Currently we have designed a lean interface for expressing the semantics of logical processes in JAMES II only comprising an event execution method and a method to return any new events.

5 SIMULATING LOGICAL PROCESSES

There is a large number of different simulation algorithms for the logical process paradigm (Fujimoto 2000). In general, these solutions can be subdivided into sequential and distributed algorithms, and the latter can be further subdivided into algorithms based on a conservative or an optimistic synchronization protocol, or a combination of both. In the following we give a brief overview, focusing on some basic algorithms which have already been realized and evaluated by others. We implemented these for our environment, and together with the provided models they form a solid foundation which can and should be extended in the future.

5.1 Sequential simulation

Sequential discrete-event simulation may often be the best way to simulate LP-based models, especially if a model is sufficiently small to fit into the memory of a single computer, if it does not take too long to be computed, or if it can be combined with a coarse-grained parallel simulation, i.e., distributing replications of a sequential simulation run over several processors. Sequential simulation means that the simulation algorithm sequentially computes the model and thus there is no need to partition it, to balance the load, or to deal with communication overhead. In addition, implementations of sequential algorithms put the results achieved by the PDES versions into perspective, and thus provide a baseline for the identification of model classes for which PDES actually pays off. A simple sequential algorithm manages all events in an event queue. It retrieves all events with a minimal time stamp and then executes them one after the other. Each event is executed by the model at first. Then, all newly generated events caused by its execution are added to the global event queue.

5.2 Conservative, distributed simulation

Conservative synchronization protocols ensure the local causality constraint by letting each processor only process those events that are *safe*, i.e., for which it is sure that no straggler event will arrive. Such protocols are well-known and have been studied intensely with respect to their implementation (Fujimoto 2000) and their theoretical properties (Nicol 1993, Korniss et al. 2003). We select the null message algorithm (Bryant 1977, Chandy and Misra 1979) and the barrier synchronization algorithm (Hennessy and Patterson 2002) as examples for conservative distributed simulation algorithms because they are relatively simple to implement and have already been examined thoroughly. The null-message algorithm used in our comparisons is based on the original Candy/Misra/Bryant edition. The simple Barrier Synchronization protocol used in our comparisons makes use of a centralized mechanism for synchronization. We also implemented the Breathing Time Bucket protocol (Steinman 1991).

5.3 Optimistic, distributed simulation

Conservative synchronization protocols face several problems. Many, e.g., the null-message protocol, require to define a *lookahead*: a guaranteed time stamp difference between a processed event and new events it causes at other LPs. Some protocols may also lead to dead-locks, i.e., situations where the simulation comes to a halt because no processor can be sure to execute the next event. To detect and resolve deadlocks costs additional processing time.

In contrast, optimistic synchronization protocols execute events without being sure that time stamp order is not violated. Hence, straggler events may occur, and they require to roll back the model to a former state, from which the computation can then be continued. Storing model states costs memory, so optimistic protocols usually have to calculate the *global virtual time* (GVT), i.e., the minimum time stamp of all unprocessed events throughout the running simulation. The GVT allows to identify model states that are not necessary anymore, so that their memory can be deallocated. The Time Warp algorithm provides a general scheme for synchronizing a distributed computation optimistically (Jefferson 1985). It can exploit a significant degree of parallelism and is deadlock-free. Also, Time Warp operates transparently to the model being simulated and does not require model-specific knowledge, such as a look-ahead. However, Time Warp has certain performance pitfalls and trade-offs. Many optimizations that enhance the original algorithm have been proposed in the literature (Fujimoto 2000). In the current implementation, we use a synchronous algorithm for GVT calculation. A `GVTController` starts GVT calculation after sleeping for a certain amount of wall clock time.

6 AN EXTENSIBLE FRAMEWORK OF LP SIMULATORS

Logical processes are independent entities that communicate via events. How these events are transferred and how processes are triggered to execute them depends on the synchronization protocol, i.e., the actual simulation algorithm. As we separate model and simulator, the simulation algorithms, realizing the different synchronization schemes, obviously need to be "exchangeable entities" – particularly, as their performance may vary greatly with the characteristics of the model they are applied to (see Section 7.3). In addition, there is a large variety of auxiliary algorithms that can be combined with the

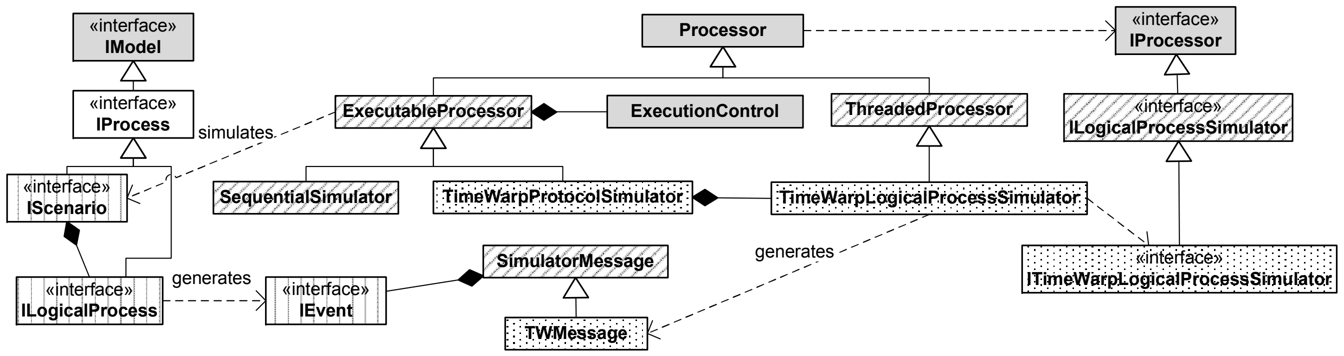

Figure 1: The general structure of the experimentation environment: Based on the JAMES II core classes and interfaces (grey), we developed separate classes and interfaces for models (horizontal lines) and backbone classes for PDES algorithms (diagonal lines). As an example, the integration of the central classes for Time Warp is outlined (dots).

simulation algorithms, e.g., event queues, partitioning methods, and algorithms for load-balancing. New classes of algorithms might be devised in the future, making it challenging to find an architecture that allows for future extensions in all these directions.

Figure 1 outlines some central entities in the experimental analysis environment. Model and simulation entities are clearly separated; simulators have access to model entities, but not vice versa. The set of LPs that defines a model is represented by `IScenario`. As JAMES II requires a processor to allow stopping, pausing, and resuming a simulation, the PDES simulators consist of two components: a *protocol simulator*, which implements all operations demanded by JAMES II, and the actual simulators for LPs, which get notified by the protocol simulator when they shall stop or resume. Much of the processor's standard execution logic has already been realized in a general `ExecutionControl` component, so that the protocol simulator merely has to implement the notification of its subordinate LP simulators. As can be seen in Figure 1, many central entities are just Java interfaces and thus do not provide any logic. This allows to exchange almost every aspect of this framework with alternative implementations – a prerequisite for extensibility in general and for the fairness of the environment in regards to comparability. The framework provides default implementations to keep the implementation effort in testing new algorithms low.

6.1 Algorithm Combinations

The algorithms to be combined with the simulation algorithms can be divided into three categories: Firstly, there are algorithms that affect simulation performance but need no direct interaction with the simulators, so that they can be hidden behind interfaces provided by the simulation system. For example, the partitioning framework of JAMES II was designed to be transparent to the actual simulation algorithms – the processor factories merely take the *results* of the partitioning step into account when instantiating the processors (Ewald, Himmelspach, and Uhrmacher 2006). Similarly, there are various ways to store observed data. Since JAMES II employs the Observer and Mediator patterns here, observation is completely decoupled from execution. Yet, both partitioning and observation may have a strong impact on overall performance.

Secondly, a simulation algorithm may delegate sub-tasks to sub-algorithms. A classic example for this are event queues. JAMES II provides several event queue implementations as plug-ins, and it could be shown that their performance is dependent on both model and simulator (Himmelspach and Uhrmacher 2007a). Extending a PDES simulator by such sub-algorithms is quite straightforward, as JAMES II provides a plug-in registry that can be queried for available algorithms (see Section 6.2). Hence, this kind of algorithm is easy to integrate, provided that the programmer of the simulator identifies all important sub-tasks and requests a plug-in for each of them from JAMES II. The interaction between simulator and sub-algorithm is controlled by the simulator, which invokes methods from the interface that comes along with the corresponding plug-in type. Thereby, new kinds of sub-algorithms can be integrated easily.

Finally, there are algorithms that interact with PDES simulators by controlling them. Load balancing (LB), for example, requires to gather load information from the processors and to select eligible parts of the model, before deciding on a new placement of model and simulator entities (Zhou 1988). Transferring entities at runtime typically involves starting and stopping the simulators, so these kinds of algorithms need to control the PDES simulators to a certain extent, and not vice versa. As an additional difficulty, some aspects of LB might be dependent on the synchronization protocol (e.g., (Carothers

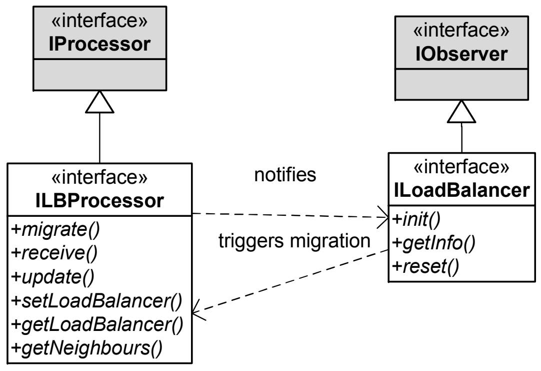

Figure 2: Relation between a PDES simulator that supports load balancing and the load balancing algorithm

and Fujimoto 1996)), while others, such as actually moving models and simulators to other machines, should be shared between all approaches. We tackled the problem of supporting future algorithms 'on top' by introducing some additional structures for LB, as depicted in Figure 2.

The basic idea is quite simple: by pre-defining the interaction between a PDES simulator and an associated algorithm, such as a load balancer, we could rely on the Observer pattern (Gamma et al. 1995, p. 293 et sqq.). A processor supporting LB could implement a specific interface, `ILBProcessor`, which implements the functions the load balancer needs for re-partitioning, i.e., the transfer of model entities. Similarly, the load balancer needs to know about neighbor simulators to which the load could be transferred. Additionally, the processor has to notify the load balancer when relevant events, such as a roll-back, happen. This allows the load balancer to continuously gather runtime statistics. While Figure 2 suggests that a load balancer is associated with *each* PDES simulator, it is important to notice that different kinds of load balancers may be employed in a simulation. For example, one setup could associate very basic load balancers to all simulators except one, which in turn collects the runtime information from the basic load balancers, decides on a re-partitioning, and then triggers the corresponding basic load balancers to start transferring specified model entities. This central LB scheme is as easy to implement as completely decentralized LB, and any kind of hybrid approach is realizable as well. Current work is dedicated towards testing this approach by implementing several standard LB approaches.

6.2 Exploiting JAMES II

JAMES II's design principle is the "Plug'n simulate" concept (Himmelspach and Uhrmacher 2007b). Most parts of the framework can be exchanged because the concrete functionality is realized as a plug-in. This concept matches the requirements of an extensible experimentation environment for combinations of algorithms quite well. An experimentation layer (Himmelspach, Ewald, and Uhrmacher 2008) supports the definition and execution of experiments in a flexible and scalable manner. A variety of different experiments, e.g., for optimization, analysis, and validation, can be executed in JAMES II. Thus, any insights gained by performance analysis of PDES algorithms can be used to tune the setup of new experiments on LP-based models. In addition, experiments on LP-based models could benefit from other exchangeable parts in JAMES II, e.g., the random number generators, the probability distributions, a.s.o. The resulting plethora of possible combinations provides the base for novel research insights by giving the opportunity to explicitly identify aspects that may have a significant influence on performance results, and by offerring means to select an algorithmic setup that performs best (Ewald, Himmelspach, and Uhrmacher 2008). The environment is sketched in Figure 3. The simulation algorithm and the model used for any simulation can be freely chosen, and thus combined. Both sets (simulation algorithms and models) can be extended by anyone – and this includes all additional algorithms and data structures as well. Figure 4 depicts the general idea of the experimentation process, and how the different parts are combined to set up an experiment.

JAMES II is mostly independent from the platform it is executed on: single-core CPUs, multi-core CPUs, and various distributed set-ups can serve as computation back-end. The possibility to reuse the different parts, i.e., algorithms and data structures, on this variety of platforms will facilitate the development and evaluation of new solutions.

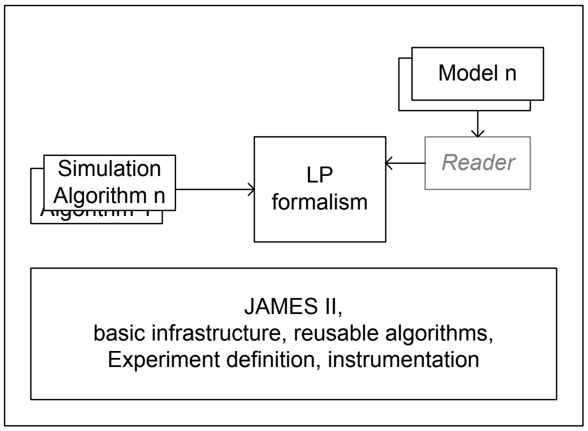
Model n
Reader
Simulation Algorithm n
LP formalism
JAMES II,
basic infrastructure, reusable algorithms,
Experiment definition, instrumentation

Figure 3: Sketch of the environment for experiments with LP based models and simulation algorithms

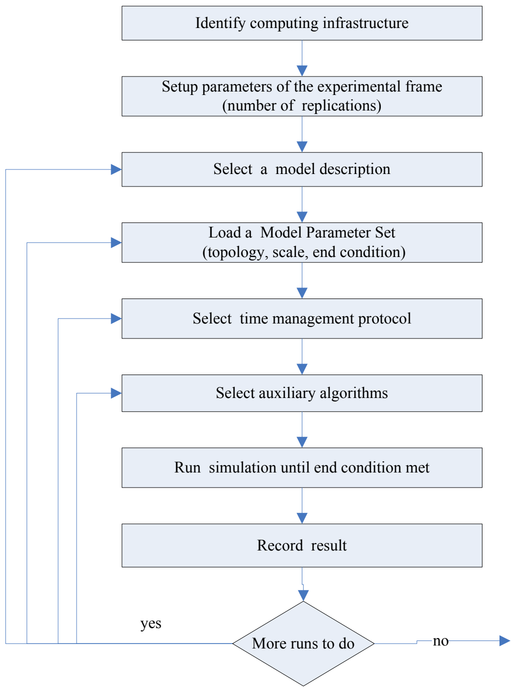
Identify computing infrastructure
Setup parameters of the experimental frame (number of replications)
Select a model description
Load a Model Parameter Set (topology, scale, end condition)
Select time management protocol
Select auxiliary algorithms
Run simulation until end condition met
Record result
More runs to do
yes
no

Figure 4: Experiment execution by using the environment. Any "loop" about sub configuration parts, e.g., certain model parameters, or auxiliary algorithms is fine.

7 EXPERIMENTAL ANALYSIS

As motivated in Section 2, a comparative analysis of algorithms has to be based on a fixed and fair experimental setup. This includes using a fixed framework, that each algorithm is realized, validated, and tuned with equal care, and that the benchmark models are designed carefully.

It is is hard to implement a set of algorithms in a comparably efficient manner, as strong optimization for one implementation introduces a serious bias to the overall comparison. Johnson suggests that *"[...] the efficiency of the code you test should be (at least to within a constant factor in practice) as good as that of a code you would expect people to use in the real world"* (Johnson 2002, p.15). We tried to realize our algorithms in this manner – not heavily optimized toward our hardware, but without serious performance bottlenecks. Our experimental results should therefore be regarded as a starting point for the comparison with additional algorithms and enhanced implementations.

7.1 Benchmark models

Suitable benchmark models (problem instances) are the cornerstone for a meaningful study on PDES algorithms. Common questions that arise in a PDES context are the *scalability* of an algorithm, i.e., the conservation of its performance characteristics when increasing problem size and number of available machines (Nicol 1998), its *speed-up* with respect to a sequential simulation, and its *(in)dependence* regarding certain model properties, e.g., granularity or interconnectedness of LPs. To investigate these aspects means to alter size and structure of the model at hand. Parameterizable benchmark models can reflect such characteristics and mimic the behavior of real-world samples. Fortunately, there are several well-known benchmark models for assessing PDES algorithm performance; we present two of them in the following. Both have been studied frequently in the last decades, and will therefore allow us to re-evaluate our results with respect to the literature. However, our approach is not limited to these two models.

PHOLD This model (Fujimoto 1990) was originally introduced to benchmark optimistic synchronization protocols. It is a parallel version of the HOLD model (Jones 1986), which is used to evaluate the performance of event queue implementations. Our experimental environment should support developers in basing their analysis on comparisons with

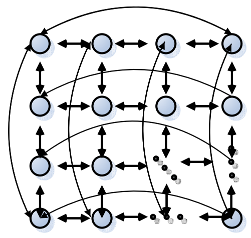

Figure 5: PHOLD toroidal grid

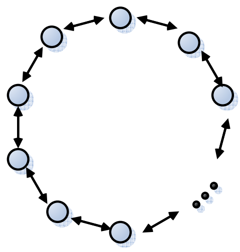

Figure 6: PHOLD ring

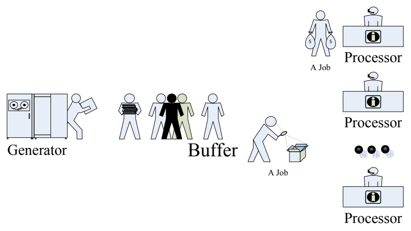

Figure 7: Generator Buffer Processor

such classic contributions, so that it is important to have an implementation of this model at hand. Although it has been introduced almost 20 years ago, it is still used in recent publications (e.g., cf. (Bauer Jr, Carothers, and Holder 2009)). A PHOLD model consists of an arbitrary number of LPs, each of which sends an event to a neighbor when an event is received. The number of events in the whole system is therefore constant and can be parameterized, as each processed event results in the generation of exactly one new event. Defining the number of events in the initial state allows to control the inherent parallelism of the model and to determine the size of the event queue in the sequential simulator (cf. (Fujimoto 1990)). Another important parameter is the *movement function*, by which the neighbor receiving the next event is selected. In the model, the neighbor is chosen randomly, with uniform probability. The *time stamp increment* defines the time difference between processed and generated event. We defined this to be normally distributed with $\mu = 4$ and $\sigma = 3$. If a negative number was drawn from this distribution, we set the time stamp increment to zero. Finally, the model allows to configure the *computational grain* by executing some synthetic load when processing an event. PHOLD is widely used as a benchmark for TimeWarp realizations, which can thereby be compared across different implementations, software architectures, and hardware platforms. For example, GEORGIATECH TimeWarp (GTW) (Das et al. 1994) and μsik (Perumalla 2005) were tested with a PHOLD model for their parallel performance. In our experiments, we used PHOLD to benchmark both conservative and optimistic synchronization protocols.

Generator/Buffer/Processor This is another well known benchmark model (e.g., cf. (Zeigler, Praehofer, and Kim 2000)), due to its simplicity and representativeness for queueing networks simulations. It is a combination of a generator LP that creates “jobs” at a certain rate, which are then buffered by a buffer LP until a processor LP is ready to execute them. There are many variations of this queuing model, e.g., with multiple generators, buffers, or processors. Its most important parameters are the probability distribution for job generation and the computational load imposed by executing a single job. We focus on a simple variant as a model for maximal throughput, by assigning a variable number or processors to a central buffer that is fed by a single generator (see Figure 7).

7.2 Experimental setup

The simulation runs have been executed on a Windows XP 64 workstation with two 2.5 GHz QuadCore Xeon Processors and 8 GB of RAM. It achieves a Java Scimark 2.0 (Java Scimark) result of 765.3 points. So far, the PDES algorithms were only tested on this multi-core machine, using a single virtual machine – instead of executing them on a set of hosts that communicate over a network connection. Hence, communication costs are low in comparison to classical PDES experiments on multiple hosts. This moves the focus of the experiments towards the question of how large the protocol-induced overhead in relation to the speed-up by using multiple cores really is – a use case that might become more and more important, as the number of cores per CPUs is likely to increase over the next years. We used JAMES II version 0.6 and Sun’s JRE 1.6.07 for 64-bit Windows.

Several PHOLD topologies were tested: $full$ denotes a complete LP graph, i.e., each LP has all other LPs as neighbors, *grid* is a two-dimensional toroidal grid (Figure 5), and in *ring* all LPs form a ring, so that each LP has only two neighbors (Figure 6). These topologies allow us to test PDES algorithm performance for tightly, medium, and sparsely connected models respectively. Moreover, we chose PHOLD model sizes from $\{4, 16, 36, 64, 100, 144, 196\}$, the number of events was set to half the model size, and a synthetic computational load was introduced. The load parameter is chosen from $k \in \{10, 20, 30\}$, and denotes $k \cdot 20000$ additional floating point operations.

The Generator/Processor/Buffer model was initialized with different amounts of processors, and hence different degrees of model-inherent parallelism.

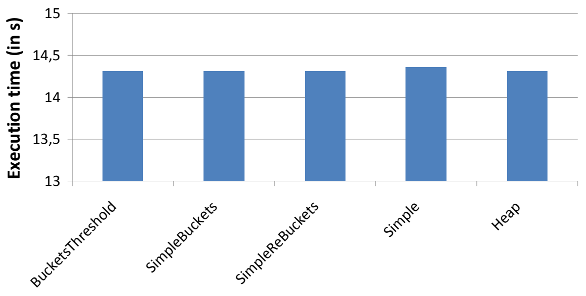

Figure 8: PHOLD performance of sequential simulation using various event queues, using a fully connected LP network with 10×10 processors

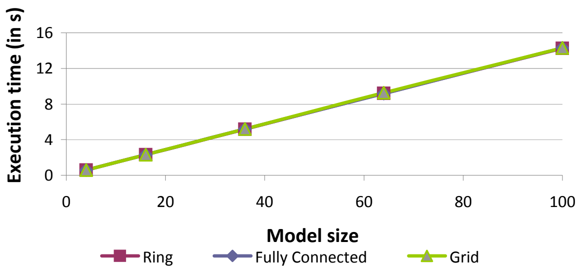

Figure 9: PHOLD performance of a sequential simulation, depending on the model size

7.3 Performance Results

At first, we evaluated the dependence of the sequential simulator performance on its central sub-algorithm, the event queue implementation. Although they play an important role in other application domains, they hardly affect the simulation runtime when executing the PHOLD model, as depicted in Figure 8. Figure 9 shows the PHOLD performance of the sequential simulator for various model sizes and the three pre-defined topologies. As expected, different topologies have no impact on the run time of the sequential simulator, and the required execution times grow linearly with the number of events to be computed.

We then explored the PHOLD performance of the parallel simulation schemes. Figure 10 illustrates the potential interdependency between synchronization protocol and model topology. While the execution times of the barrier synchronization protocol are almost identical for all three topologies, the performance of null message synchronization clearly depends on the interconnectedness of the LPs. In case of a fully connected graph, its performance suffers from the overhead of sending null messages to all neighbors. Figure 11 compares the PHOLD performance of two PDES simulators, BTB and barrier synchronization, to that of the sequential simulator. While it is not surprising that the PDES simulators are able to exploit more of the system's resources and therefore outperform the sequential simulator, the figure also suggests that the *relation* of the PDES execution times is depending on the model size – the difference between barrier synchronization and BTB is much larger for a 64-LPs model than it is for an 196-LPs model. One reason for this might be the damping of performance differences by increased threading overhead. In a last PHOLD experiment, we added different amounts of synthetic load to the model, so that the granularity was increased. As shown in Figure 12, the sequential simulator behaved as expected again, since it exhibited a linear growth in execution time. Direct comparison with BTB, barrier synchronization, and TimeWarp once again confirms that a high granularity has a positive impact on PDES speed-up – a common finding in PDES studies, e.g., in (Wonnacott and Bruce 1996).

Finally, Figure 13 presents PDES performance for the Generator/Buffer/Processor model, for different numbers of processors. These results were surprising, as they did not only show a large advantage for the sequential simulator – which could be expected, due to the low model-inherent parallelism – but also that the performance characteristics between the PDES algorithms differ greatly. The conservative barrier synchronization approach is not able to exploit the inherent parallelism at all, as indicated by the linear growth of execution time (due to increased synchronization overhead for larger models). However, TimeWarp execution times *decrease* with model size, which led us to the conclusion that this particular model setup is quite suitable for optimistic protocols, i.e., very few roll-backs occur. The modeled processors, which work concurrently in the model, are independent of each other, so that an optimistic PDES approach has huge advantages.

All in all, these examples suggest that the performance behavior of PDES algorithms is by no means trivial and highly problem-dependent – which again motivates the development of a clean environment for experimental analysis.

8 CONCLUSIONS

The benefits of the plug'n simulate concept in JAMES II have been demonstrated before, e.g., in the context of event queues (Himmelspach and Uhrmacher 2007a), simulators (Himmelspach et al. 2007), and partitioning schemes (Ewald, Himmelspach, and Uhrmacher 2006). By supporting the logical process metaphor, we broaden the scope of JAMES II and address the particularly challenging field of evaluating parallel and distributed discrete-event simulation algorithms. First

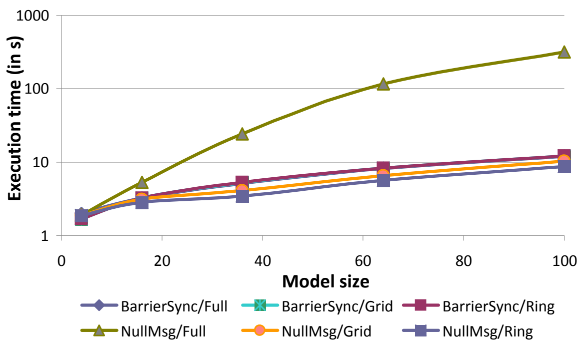

Figure 10: Performance of barrier synchronization and the null message protocol on different PHOLD topologies. The time stamp increment in this setup was set to a fixed value (`2.0`), so that the null message protocol could cope with it.

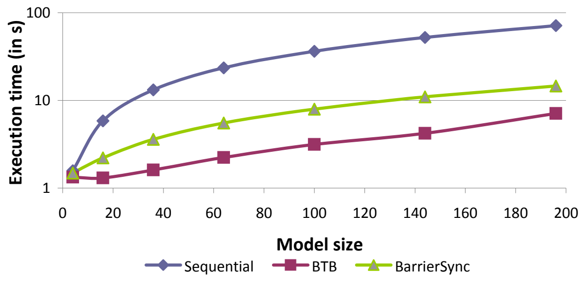

Figure 11: PHOLD Performance of BTB and barrier synchronization, in comparison to a sequential simulation

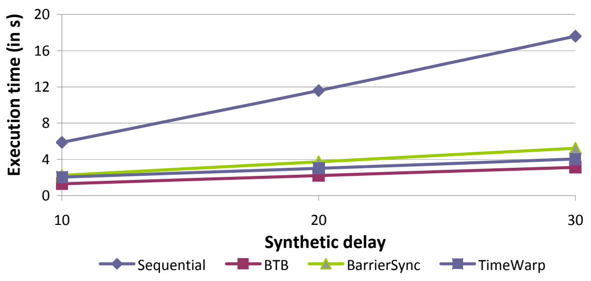

Figure 12: PHOLD performance with varying synthetic delay

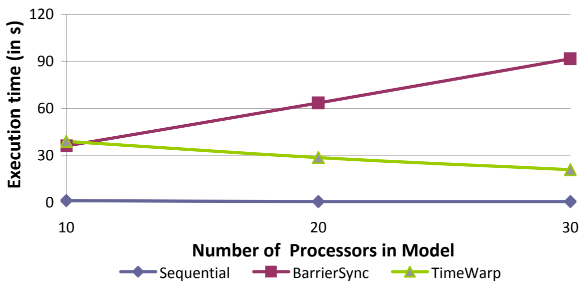

Figure 13: PDES performance for Generator-Buffer-Processor model

evaluation results for well-known algorithms have been presented, which not only confirmed what is already known but also provided some surprising insights (see discussion on G/B/P performance in section 7.3). However, the main objective of the endeavor has been to introduce an environment for the development and experimental analysis of algorithms for the logical process paradigm. The focus of future research will be to develop and evaluate further simulation algorithms. The interplay of the simulation algorithms with other algorithms and data structures, e.g., partitioning and load-balancing schemes, shall be explored carefully. The insights, gained by using current hardware such as multi-core CPUs, might lead to new efficient and well-evaluated solutions. We hope that the presented architecture contributes to these developments by helping researchers to soundly evaluate their solutions: by using the same benchmark model implementations, by considering different infrastructures, and by keeping everything except the suggested improvement fixed.

The performance database of JAMES II will allow us to capture essential details of these performance studies and their experimental set-ups on a large scale. This could foster data analyses that yield new perspectives on the performance trade-offs among distributed simulation algorithms (Ewald, Himmelspach, and Uhrmacher 2008). For the framework to grow into a research resource for distributed simulation algorithms, potential evaluation schemes, and performance data alike, we invite other people to join us: to repeat our findings, to add new benchmark models and algorithms, and to evaluate those. The JAMES II framework is available at `http://www.jamesii.org`. It is distributed under a dual license, allowing open source and proprietary usage.

9 ACKNOWLEDGMENTS

This work has been funded by the German research foundation (DFG), the *China Scholarship Council*, the *National Science Foundation of China* (No. 60773019), and the *National High-Tech Research and Development Plan of China* (No. 2006AA01Z330). Bing Wang was visiting the Modeling and Simulation Research Group at the University of Rostock while this work was undertaken.

REFERENCES

1998. JWarp: A java library for parallel discrete-event simulations. In *Poster Paper at ACM Workshop on Java for High-Performance Network Computing*, 10–11.

Bauer Jr, D. W., C. D. Carothers, and A. Holder. 2009, June. Scalable time warp on blue gene supercomputers. In *2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 35–44. IEEE: IEEE CPS.

Bryant, R. E. 1977. Simulation of packet communication architecture computer systems. Technical report, Cambridge, MA, USA.

Carothers, C. D., and R. M. Fujimoto. 1996. Background execution of time warp programs. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, 12–19. Washinton: IEEE Computer Society Press.

Chandy, K. M., and J. Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *Software Engineering, IEEE Transactions on* SE-5 (5): 440–452.

Chen, G., and B. K. Szymanski. 2005, December. Dsim: scaling time warp to 1,033 processors. In *Proceedings of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 346–355: Winter Simulation Conference.

Cowie, J. 1998. Parallel discrete-event simulation in java. In *In Proceedings of ACM 1998 Workshop on Java for High Performance Netwrok Computing*, 251–254.

Curry, R., C. Kiddle, R. Simmonds, and B. Unger. 2005. Sequential performance of asynchronous conservative pdes algorithms. In *PADS '05: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, 217–226. Washington, DC, USA: IEEE Computer Society.

Das, S. R., R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, Proceedings of the 1994 Winter Simulation Conference. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Ewald, R., J. Himmelspach, and A. M. Uhrmacher. 2006. A non-fragmenting partitioning algorithm for hierarchical models. In *Proceedings of the 2006 Winter Simulation Conference*, ed. L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 848–855. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Ewald, R., J. Himmelspach, and A. M. Uhrmacher. 2008. An algorithm selection approach for simulation systems. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS) 2008*, Volume 22, 91–98. Los Alamitos, CA, USA: IEEE Computer Society.

Ferscha, A., J. Johnson, and S. J. Turner. 2001, September. Distributed simulation performance data mining. *Future Generation Computer Systems* 18 (1): 157–174.

Ferscha, A., and M. Richter. 1997, December. Java based conservative distributed simulation. In *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. J. Healy, and D. H. Withers, 381–388. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Fujimoto, R. M. 1990. Performance of Time Warp under synthetic workloads. In *Proc. of the SCS Multiconf. on Distributed Simulation*, 23–28.

Fujimoto, R. M. 2000. *Parallel and distributed simulation systems*. John Wiley and Sons.

Fujimoto, R. M., and M. Hybinette. 1997. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul.* 7 (4): 425–446.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns*. Addison Wesley.

Gent, I. P., S. A. Grant, E. MacIntyre, P. Prosser, P. Shaw, B. M. Smith, and T. Walsh. 1997, May. How not to do it. Technical report, University of Leeds.

Goes, L. F. W., L. E. S. Ramos, and C. A. P. S. Martins. 2004. Clustersim: a java-based parallel discrete-event simulation tool for cluster computing. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 401–410. Washington, DC, USA: IEEE Computer Society.

Gupta, A., I. F. Akyldiz, and R. M. Fujimoto. 1991, October. Performance Analysis of Time Warp With Multiple Homogeneous Processors. *IEEE Trans. on Softw. Eng., Special Section on Parallel Systems Performance* 17 (10): 1013–1027.

Hendrickson, B., and T. G. Kolda. 2000, November. Graph partitioning models for parallel computing. *Parallel Comput.* 26 (12): 1519–1534.

Hennessy, J. L., and D. A. Patterson. 2002, May. *Computer architecture: A quantitative approach (the morgan kaufmann series in computer architecture and design)*. Morgan Kaufmann.

Himmelspach, J., R. Ewald, S. Leye, and A. M. Uhrmacher. 2007. Parallel and distributed simulation of parallel devs models. In *Proceedings of the SpringSim '07, DEVS Integrative M&S Symposium*, 249–256: SCS.

Himmelspach, J., R. Ewald, and A. M. Uhrmacher. 2008. A flexible and scalable experimentation layer. In *Proceedings of the 2008 Winter Simulation Conference*, ed. S. J. Mason, R. R. Hill, L. Moench, O. Rose, T. Jefferson, and J. W. Fowler, 827–835. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Himmelspach, J., and A. M. Uhrmacher. 2007a. The event queue problem and pdevs. In *Proceedings of the SpringSim '07, DEVS Integrative M&S Symposium*, 257–264: SCS.

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

Java Scimark. http://math.nist.gov/scimark2/.

Jefferson, D., B. Beckman, F. Wieland, L. Blume, and M. Diloreto. 1987. Time warp operating system. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, 77–93. New York, NY, USA: ACM.

Jefferson, D. R. 1985. Virtual time. *ACM Trans. Program. Lang. Syst.* 7 (3): 404–425.

Jha, V., and R. L. Bagrodia. 1994. A unified framework for conservative and optimistic distributed simulation. In *PADS '94: Proceedings of the eighth workshop on Parallel and distributed simulation*, 12–19. New York, NY, USA: ACM.

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

Jones, D. W. 1986, April. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM* 29 (4): 300–311.

Korniss, G., M. A. Novotny, H. Guclu, Z. Toroczkai, and P. A. Rikvold. 2003. Suppressing roughness of virtual times in parallel discrete-event simulations. *Science* 299 (5607): 677–679.

LaMarca, A., and R. E. Ladner. 1997. The influence of caches on the performance of sorting. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, 370–379. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Leye, S., J. Himmelspach, M. Jeschke, R. Ewald, and A. M. Uhrmacher. 2008. A grid-inspired mechanism for coarse-grained experiment execution. In *Proceedings of the 12th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, ed. A. E. S. David Roberts and A. Ferscha, 7–16. IEEE Computer Society: IEEE Press.

Martin, D. E., T. McBrayer, and P. A. Wilsey. 1996. Warped: a time warp simulation kernel for analysis and application development. In *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, ed. E. H. Rewini and B. D. Shriver, Volume 1, 383–386.

Martin, D. E., P. A. Wilsey, R. J. Hoekstra, E. R. Keiter, S. A. Hutchinson, T. V. Russo, and L. J. Waters. 2003. Redesigning the WARPED Simulation Kernel for Analysis and Application Development. In *Annual Simulation Symposium*, 216–223.

McGeoch, C. 2001, March. Experimental analysis of algorithms. *Notices of the AMS* 48 (3): 304–311.

McGeoch, C. 2007, November. Experimental algorithmics. *Comm. of the ACM* 50 (11): 27–31.

Misra, J. 1986, March. Distributed distcrete-event simulation. *ACM Computing Surveys* 18 (1): 39–65.

Nicol, D. M. 1993. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM* 40 (2): 304–333.

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., M. M. Johnson, A. S. Yoshimura, and D. O. C. Science. 1998. The infrastructure for distributed enterprise simulation.

Perumalla, K. 2005, June. μsik: A micro-kernel for parallel/distributed simulation systems. In *Workshop on Parallel and Distributed Simulation*, 59–68. Monterey, CA: IEEE.

Perumalla, K. S. 2006. Parallel and distributed simulation: traditional techniques and recent advances. In *Proceedings of the 2006 Winter Simulation Conference*, ed. L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 84–95. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Perumalla, K. S. 2007. Scaling time warp-based discrete event execution to 104 processors on a blue gene supercomputer. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, 69–76. New York, NY, USA: ACM.

Peschlow, P., M. Geuer, and P. Martini. 2008, April. Logical process based sequential simulation cloning. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, 237–244.

Smith, J. S., J. A. Hamilton, R. E. Nance, B. L. Nelson, G. F. Riley, and L. W. Schruben. 2008. Panel discussion: What makes good research in modeling and simulation: Assessing the quality, success, and utility of M&S research. In *Proceedings of the 2008 Winter Simulation Conference*, ed. S. J. Mason, R. R. Hill, L. Moench, O. Rose, T. Jefferson, and J. W. Fowler, 689–694. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Steinman, J. S. 1991, January. SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, 95–101.

Teo, Y. M., and Y. K. Ng. 2002. Spades/java: Object-oriented parallel discrete-event simulation. In *SS '02: Proceedings of the 35th Annual Simulation Symposium*, 245. Washington, DC, USA: IEEE Computer Society.

Wonnacott, P., and D. Bruce. 1996. The APOSTLE simulation language: granularity control and performance data. In *PADS '96: Proceedings of the tenth workshop on Parallel and distributed simulation*, 114–123. Washington, DC, USA: IEEE Computer Society.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. Academic Press.

Zhou, S. 1988, September. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Transactions on Software Engineering* 14 (9): 1327–1341.

AUTHOR BIOGRAPHIES

BING WANG is a Ph.D. candidate at School of Computer Science, National University of Defense Technology, P.R. China. In this school, He received a B.S. and a M.S. degree in Computer Science in 2003 and 2006 respectively. His current research interests are parallel discrete event simulation, modeling methodology for complex systems, and experimental simulation algorithm evaluation. He has visited the MoSi Group in University of Rostock, for one year. He can be contacted by email at <wangbing (at) nudt.edu.cn>.

JAN HIMMELSPACH is a post doc in the Computer Science Department at the University of Rostock. He received his doctorate in Computer Science from the University of Rostock. His research interest is on software engineering for modeling and simulation, credibility of modeling and simulation, and on efficient modeling and simulation solutions. His e-mail address is <jan.himmelspach (at) uni-rostock.de>

ROLAND EWALD holds a diploma in Computer Science from the University of Rostock and pursues a PhD at the Modeling and Simulation Group at the University of Rostock. His main research interests are in simulation algorithm selection and performance analysis.
His e-mail address is <roland.ewald (at) uni-rostock.de>.

YIPING YAO is a professor in School of Computer Science, National University of Defense Technology, R.R. China. In this school, he received his M.S. and Ph.D. degrees in 1987 and 2004 respectively. He received his B.S. degree in computer science from Huazhong University of Science and Technology in 1985. At present, he has achieved 2 second-class National Science and Technology Advance Awards and 8 Provincial Science and Technology Advance Awards. His research interests include modeling methodology for complex systems, high performance simulation, distributed simulation, and virtual reality. His e-mail address is <ypyao (at) nudt.edu.cn>.

ADELINDE M. UHRMACHER is Professor at the Department of Computer Science at the University of Rostock and head of the Modeling and Simulation Group. She received her doctorate in Computer Science from the University of Koblenz-Landau. Her research interests are in modeling and simulation methodologies and their applications. Her e-mail address is <adelinde.uhrmacher (at) uni-rostock.de>