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

AUTOMATING THE RUNTIME PERFORMANCE EVALUATION OF SIMULATION ALGORITHMS

Roland Ewald
Adelinde M. Uhrmacher

Institute of Computer Science
University of Rostock
18059 Rostock, Germany

ABSTRACT

Simulation algorithm implementations are usually evaluated by experimental performance analysis. To conduct such studies is a challenging and time-consuming task, as various impact factors have to be controlled and the resulting algorithm performance needs to be analyzed. This problem is aggravated when it comes to comparing many alternative implementations for a multitude of benchmark model setups. We present an architecture that supports the automated execution of performance evaluation experiments on several levels. Desirable benchmark model properties are motivated, and the quasi-steady state property of such models is exploited for simulation end time calibration, a simple technique to save computational effort in simulator performance comparisons. The overall mechanism is quite flexible and can be easily adapted to the various requirements that different kinds of performance studies impose. It is able to speed up performance experiments significantly, which is shown by a simple performance study.

1 INTRODUCTION

Even after several decades of research in simulation algorithms, it still requires much effort to thoroughly analyze the performance of a newly developed simulation algorithm. One reason for this might be that the notion of performance has many facets: users are not only concerned with execution speed, but also with memory consumption or the accuracy of the obtained results. In many cases, a delicate trade-off between these requirements has to be found. It is also mandatory to compare the performance of a new implementation to alternative solutions. This is of particular importance in research, as the relation of a new approach to existing ones allows to show the scientific contribution, a major issue the M&S community is currently discussing (Smith et al. 2008).

Both aspects of the problem can be alleviated to some extent: the fair comparison of algorithms can be achieved by implementing them (equally well) in the same language and running them on the same hardware and benchmark models, thereby allowing them to re-use the same data structures and sub-algorithms, e. g., event queues. Comparative studies are particularly well-suited for extensible open source simulation systems such as JAMES II (Himmelspach and Uhrmacher 2007b), a Java-based framework for which the presented mechanisms have been implemented. Such tools allow to freely combine existing code with new implementations, and hence to test every new component against all others.

On the other hand, the ability to freely combine algorithms may lead to a combinatorial explosion. For example, there are scenarios where flexible simulation tools like JAMES II offer virtually *thousands* of algorithms, e. g., combinations of random number generator, event queue, parallel discrete-event simulator, and partitioning algorithm: having only ten implementations for each component would still result in 10^5 algorithm combinations! This motivates large-scale performance experiments, and consequently leads to an overabundance of algorithm performance data. To analyze it conveniently, the data could be written to a dedicated performance database, followed by a semi-automated analysis, e. g., by data mining methods.

However, there is a *third* aspect: even if there is a set of comparable simulation algorithms and there are means to store their performance data and analyze it afterward, how should the performance comparison *experiment* be designed? This task is usually done manually – which is not necessarily bad, as the developer often knows intuitively which specific properties of the algorithm have to be investigated with particular rigor. Still, when faced with dozens or hundreds of competing algorithms or benchmark models that are highly parameterizable, a manual experiment definition can be very time-consuming and may lead to inconclusive results due to insufficient data. Some performance studies even seem quite impossible without some degree of automation, as illustrated in section 6.

Several techniques can be combined to tackle this issue. The problem of designing *sequential* experiments, i. e., experiments which will be adjusted during their execution by considering intermediate results, has a long tradition in statistics (e. g., cf. (Robbins 1952)). Such experiment design approaches have to be combined with techniques from *experimental algorithmics* (McGeoch 2007), a field that is concerned with the empirical analysis of algorithms in general. Finally, the context of simulation allows us to deal with some aspects of the overall problem in a more specific manner, e. g., when it comes to define suitable problem instances for benchmarking.

2 BACKGROUND AND RELATED WORK

A fundamental approach to algorithm comparison is complexity theory, as it highlights the essential differences between alternative solutions and thereby supports the development of new algorithms. Nevertheless, classical complexity theory is insufficient to characterize overall algorithm performance with a level of detail that is satisfactory for all practical considerations. For example, specific properties of the given input instance (Cheeseman, Kanefsky, and Taylor 1991) or hardware (LaMarca and Ladner 1997) may have a high impact, which is generally hard to predict, quantify, or formalize. This led to some advancements in complexity theory (Fellows 2001), but also to the adoption of a more empirical perspective on algorithm performance (Johnson 2003). Experimentation with algorithms can be facilitated by various techniques, but there are also many pitfalls that must be avoided to obtain unbiased and meaningful results.

An important issue in performance studies is the design of representative input data. It is usually *not* sufficient to merely consider some randomly created instances, since *"'[...] real inputs are not random, but rather have lots of hidden structure [...]"'* (Fellows 2001, p. 298). If such structures are not taken into account, the comparison is likely to be biased towards algorithms that perform well under circumstances that rarely occur.

Taking advantage of *structured* random instances, however, lets the experimenter exploit their flexibility regarding size and other parameters (see discussion in (Johnson 2003) and section 3). Examples for structured random benchmark instances can be found in many fields, e. g., compiler optimization (Yu, Zhang, and Rauchwerger 2004) or theoretical computer science (Gagliolo and Schmidhuber 2006), where standardized problem libraries are used to test heuristics for hard problems. The compilation of such libraries is greatly facilitated by additional knowledge on algorithmic challenges, e. g., the existence of *phase transitions* (Cheeseman, Kanefsky, and Taylor 1991, Hogg, Huberman, and Williams 1996), although this focus on particularly hard problem instances may also divert the research focus to rather unrealistic scenarios (Johnson 2003). For simulation studies, benchmark models are either built for specific applications, e. g., (Cao, Li, and Petzold 2004), for specific simulation algorithms, e. g., (Fujimoto 1990), or for specific formalisms like DEVS (Glinsky and Wainer 2005).

If a set of benchmark input is defined, e. g., in form of a set $\mathbb{B} = \{b_1, \ldots, b_n\}$ that shall be tested on a set of computer systems $\mathbb{C} = \{c_1, \ldots, c_m\}$, a *performance space* of the algorithm set $\mathbb{A} = \{a_1, \ldots, a_k\}$ can be defined as $\mathbb{P} = \mathbb{A} \times \mathbb{B} \times \mathbb{C}$. The general goal of performance studies is to associate every element of $\mathbb{P}$ with an element of $\mathbb{R}^x$, a vector that contains all desired performance information, e. g., mean and standard deviation of the required execution time. This simplified formalization resembles that of (Rice 1976), who formally defined the *algorithm selection problem* on similar concepts. The key idea behind algorithm selection is to provide a user with means to *automatically* select a suitable algorithm for a given problem, and hence requires the analysis of past performance data, e. g., cf. (Ramakrishnan, Rice, and Houstis 2002).

It is also in this context that *"'[...] the process of searching itself as an interesting and challenging problem"'* (Vuduc, Demmel, and Bilmes 2001, p. 125) has been subject to research. This is because the performance space $\mathbb{P}$ is usually much too large to simply enumerate it and execute each configuration one-by-one. Moreover, runtime performance metrics may vary from execution to execution, so that replications, i. e., repeated executions of an algorithm on the same input data, are necessary to obtain statistically significant results. This problem is of particular importance when studying simulation algorithms, since many models contain stochastic elements and hence impose *varying* workloads per replication. There are many variance reduction techniques that could alleviate the problem (McGeoch 1992), but applying them can also be counterproductive – statistically (Ankenman, Nelson, and Staum 2008), and because a *realistic* variance estimation might be of interest as well. Besides stochastic models, there are also stochastic simulation algorithms, e. g., in Computational Biology, where performance analyses face the same problem (Jeschke and Ewald 2008). This motivates the investigation of performance space exploration techniques that are tailored to the field of simulation algorithms.

For the *empirical tuning* of compilers, (Vuduc, Demmel, and Bilmes 2001) introduced a stopping rule for a random search in $\mathbb{A}$, where $|\mathbb{B}| = |\mathbb{C}| = 1$. It estimates the probability that the currently best known algorithm does belong to the overall best ε percent of algorithms in $\mathbb{A}$: the user can now stop the random search in $\mathbb{A}$ automatically when, for example, the probability that the best found algorithm belongs to the top $\varepsilon = 10\%$ is above 95%.

For smaller sets of algorithms, it may suffice to conduct factorization experiments, which allow to assess the sensitivity of investigated parameters and their interactions. (Yu, Zhang, and Rauchwerger 2004) use such a setup for evaluating different reduction algorithms for parallel computing. (Ferscha, Johnson, and Turner 2001) have done similar experiments for some parallel and distributed discrete-event simulation algorithms. Their factorization experiment included, besides various algorithmic variables, different execution platforms and models. However, a full factorization with n parameters requires 2^n simulation setups, each of which usually has to be replicated. Even if the number of resulting simulation runs is feasible or was reduced by any of the classical approaches like (Plackett and Burman 1946), only two values per parameter have been evaluated, which might be unsatisfying, e. g., for scalability studies. The field of experiment design, which aims at improving the efficiency and validity of experiments by considering more sophisticated statistical methods, originated from the work of Fisher in the 1920s (Box 1980). Today, advanced statistical models can be used to steer experiments into interesting regions, e. g., (Ankenman, Nelson, and Staum 2008) employ stochastic kriging for meta-modeling and thereby identify the regions in the parameter space that will yield most surplus in statistical terms. Such meta-modeling methods could also steer performance experiments through unknown parameter spaces.

Another interesting sub-field of experiment design is concerned with the so-called *multi-armed bandit problem*, which is a sequential experiment design problem (Robbins 1952) and relates to various real-world applications, e. g., clinical trials (Jun 2004). In its most simple form, the task is to design an experiment that extracts the maximal reward from a gambling machine with two arms within x rounds. For each round it has to be decided which arm to pull, afterwards the experimenter receives a random reward from the chosen arm's unknown reward distribution. Although optimal solutions exist for specific cases (Gittins 1989), various heuristics have been proposed and evaluated in more general settings (Auer, Cesa-Bianchi, and Fischer 2002, Vermorel and Mohri 2005). None of the aforementioned methods has been specifically developed for algorithmic performance analysis, but almost all of them are applicable to the problem in some manner.

Which of the aforementioned experiment design techniques – (fractional) factorization, kriging, or bandit policies – is the most effective? This strongly depends on the nature of the study (see section 5.1). We believe that the bevy of existing methods requires a structured and extensible mechanism that makes sophisticated experimental techniques available to all simulationists concerned with performance.

3 BENCHMARK MODELING FOR SIMULATOR PERFORMANCE ANALYSIS

3.1 Requirements

Using benchmark models for performance evaluation is commonplace in the field of simulation – however, there seems to be little guidance from the modeling and simulation literature about how to create them: What makes a model a good performance benchmark model? How do the properties of a benchmark model reflect its purposes?

The performance evaluation community mainly distinguishes between *macro-* and *micro*-benchmarks. Micro-benchmarks are small and relatively simple programs that are constructed for analyzing a particular aspect of the system under study (Small et al. 1997). Macro-benchmarks subsume real-world *applications* or application *kernels* (i. e., their most characteristic parts), but also recorded *traces* of application executions and *synthetic* benchmarks (Agrawal et al. 2008, Small et al. 1997). Synthetic macro-benchmarks are constructed to mimic relevant aspects of real-world applications. They can, e. g., be derived from micro-benchmarks (Agrawal, Arpaci Dusseau, and Arpaci Dusseau 2008) and are useful for exploring different realistic scenarios by changing parameters (Skadron et al. 2003). Some of these benchmark types can be translated to simulator performance evaluation: real-world models can be regarded as application benchmarks, whereas simple benchmark models to determine the cost of certain operations can be regarded as micro-benchmarks, e. g., to study the performance impact of roll-backs in an optimistic parallel discrete-event simulation. For evaluating the overall performance of a simulation algorithm, parameterizable synthetic benchmark models should usually be preferred over application benchmarks. This is because application benchmarks often lack some important properties that makes them ill-suited for automated performance studies (Jeschke and Ewald 2008):

Parameterization Application benchmarks have parameters to adjust them to all scenarios of interest. These parameters usually affect a model's behavior and therefore also the performance of the simulation algorithm – but only indirectly. In contrast, the parameters of synthetic benchmark models should be specifically designed to explore all possible behaviors that models of the given type can possibly exhibit. Although this might make the parameter space very large, the correlation between parameters and algorithm performance could be stronger and more straightforward, and so will be the findings of

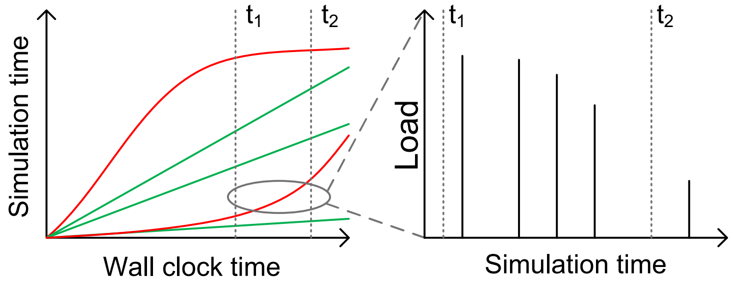

Figure 1: Models need to be in a steady state (w.r.t. computational load) if the wall-clock time shall grow linearly with the simulation time interval. If not (see red trajectories in left plot), this means that the computational load of the model changes over time (right plot).

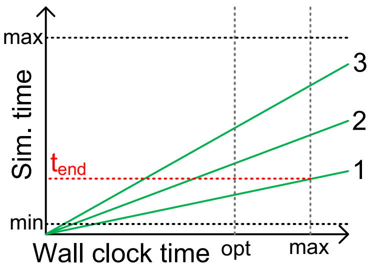

Figure 2: Calibration means to identify a simulation end time so that algorithms of differing speed (1: slowest, 3: fastest) finish execution after $\approx t^{wct}_{opt}$ all clock time

the performance study. Moreover, well-chosen benchmark model parameters may allow to identify the *features* of all models a simulation algorithm works well on, e. g., for the later application of algorithm selection techniques.

Scalability Application benchmarks are often not scalable enough for thorough performance studies – a problem that is closely related to the parameterization issue. This is of particular importance when studying new algorithms for parallel and distributed simulation, as these may only show advantageous performance for models that are large enough. Moreover, some interesting phenomena, e. g., the effects of caches (LaMarca and Ladner 1997), do not occur when testing models that are too small. Automated performance analysis could address scalability studies in various ways, e. g., by providing simple means for setting up the experiments, or for statistically analyzing the growth of the measurements under scrutiny.

Simplicity and comparability Real-world application models tend to be rather complex and intricate. In contrast, a synthetic benchmark model should be *as simple as possible*. This reduces the error potential when implementing or specifying the model for the setup at hand, and one might also gain some analytical insights into its behavior, or can facilitate debugging, simulator validation, and the interpretation of the performance experiment's results. However, the major benefit of simplicity is comparability, which also motivates the use of simple and established benchmark models. It allows to compare performance across different simulation systems, hardware platforms, or single algorithms. To ensure that the benchmarking results are indeed comparable, it is best to keep the benchmarking models simple.

Quasi-steady state Another desirable property of benchmark models is quasi-steady state behavior with respect to the characteristics of the computational load it imposes on the simulator in a given simulation time interval. Application models often have warm-up phases and therefore do not comply to this property, so that simulation time does *not* grow linearly with wall-clock time – there even might be *different* phases in model execution that are advantageous for *different* simulators. The arising uncertainty about the current state of the model, and hence the workload a simulation algorithm is confronted with, is a strong source of variance in experiments regarding execution times. Since workload characteristics and their impact on simulator performance are of particular interest in many studies, this issue should rather be investigated explicitly. Therefore, it is advantageous to construct models that remain in very similar states throughout the execution, which is often bought with some additional parameters, i. e., a higher-dimensional instance space $\mathbb{B}$ that requires more efforts to explore. However, steady-state behavior also facilitates the usage of automatic mechanisms for simulation time *calibration* (see section 4), which may speed up performance analysis significantly (see section 6.2). Note that the state only needs to be relatively steady with respect to the computational load characteristics, as depicted in Figure 1. For benchmark models without such a quasi-steady state, the time at which the simulation is stopped affects which algorithm is deemed to be the fastest. If, for example, two discrete-event simulators are compared, one might work best when encountering relatively few but complex events, e. g., until t_1 is reached (Figure 1). Afterwards, the load characteristics of the model change and at time t_2 the second algorithm might be faster. In such situations, execution time comparisons are essentially futile.

3.2 Example: PHOLD

As an example for the outlined benchmark modeling requirements, consider the PHOLD model (Fujimoto 1990), a widely used synthetic benchmark for parallel and distributed discrete-event simulators (Bauer Jr, Carothers, and Holder 2009). Its compliance with the four design principles (parameterization, scalability, simplicity, and quasi-steady state) might explain its widespread use. It is derived from the HOLD benchmark for event queues (Jones 1986) and indeed very *simple* to implement: basically, a set of model entities exchange timestamped events at random. The total number of events in the system is fixed.

If a model entity receives an event, it creates a new event with a future time stamp and sends it to a randomly chosen neighbor. The model is also *scalable* in at least two dimensions: increasing the number of events will increase the amount of model-inherent parallelism, so that model parallelism can be set into direct relation to the performance of parallel and distributed simulators. Increasing the number of model entities increases the memory footprint of the model. PHOLD is *parameterizable* in many other aspects: the topology of the model can be adjusted, i. e., which model has which neighbors, as well as the probability distribution to create future event time stamps or the synthetic computational load each event imposes on the simulator.

Very much like application models, synthetic benchmark models have to be used with care. Before conducting any serious study, it has to be analyzed if the parameters of the model actually allow to investigate the aspects of interest. For example, PHOLD as such does not lend itself to studies on dynamic load balancing algorithms, as it also fulfills the *quasi-steady state* requirement: the computational load of the model does not change over time. It is always the same number of events that is propagated through the same neighborhood. Consequently, PHOLD needs to be adjusted when algorithms that rely on exactly this kind of model dynamics, such as load balancers, shall be studied. For example, a subset of nodes with additional behavior can be added (Low 2002). Note that the benchmark model may retain the quasi-steady state behavior for larger *intervals* of simulation time, e. g., by controlling model dynamism with fixed parameters for frequency and amplitude of load changes.

Finally, it should be noted that all performance studies, however interesting their results may be, are worth nothing unless the benchmark input space $\mathbb{B}$ is chosen so that it *represents* real-world problems. In other words, the benchmark model parameter space has at least to include the model size, structure, and behavior that can also be found in comparable application models. This is hard to achieve and usually involves application-specific surveys, as well as conjectures regarding future model properties. Representativeness is nevertheless crucial to the interpretation of the results, e. g., when it comes to deciding which algorithm scales better and should therefore be ported to a cluster machine – consequently, lack of representativeness is an often raised criticism of performance analyses (Gagliolo and Schmidhuber 2006, Johnson 2003, Small et al. 1997). Ideally, one would address this problem by additional application-based benchmarking for the validation of the general findings.

4 SIMULATION TIME INTERVAL CALIBRATION

This section describes how the simulation end time can be calibrated for a given benchmark model, in order to speed up experiments concerned with execution speed. Calibration refers to the process of setting the time interval for which a given benchmark model shall be simulated. Without loss of generality, we assume that a benchmark model is executed from simulation time 0 to t_{end}^{sim}. The general idea behind simulation time interval calibration is to find a suitably small t_{end}^{sim} for a newly parameterized instance of a benchmark model, e. g., by considering similar parameterizations and the t_{end}^{sim} values that were found for them. The point behind all this is that good benchmark models in the sense of section 3.1 are scalable and have parameters that strongly impact the computational workload a simulator is faced with. Consequently, the computational loads of the elements in $\mathbb{B}$ will vary strongly, and with it the wall clock time it takes to execute them. An execution speed comparison requires fairness, i. e., all algorithms should be tested on the same model and for the same time interval – but that does not mean that this interval has to be *fixed* for all benchmark model setups, i. e., for all parameterizations of the model. It is the task of a calibration algorithm to find a suitable t_{end}^{sim} for a new benchmark model parameterization.

What does 'suitable' mean in this context? First of all, the simulation time interval should not be too small, it should be large enough to ensure a representative quasi-steady state behavior of the overall model (as discussed for PHOLD and load balancing in section 3.2). This lower simulation time limit, t_{min}^{sim}, is complemented by an upper simulation time limit t_{max}^{sim}. Why not just always choose the smallest simulation time interval, which is $[0, t_{min}^{sim}]$? While this would speed up the performance study as much as possible, the results of the study would suffer from a potentially large bias when the exploration considers elements from $\mathbb{B}$ that are relatively easy to simulate: important runtime aspects might not be recognizable anymore and unobserved variables – i. e., state of the hardware, the operating system, and the concurrently executed programs, such as virus scanners – have a large impact that may change results dramatically. This is why the end time needs to be adapted *automatically* – for small-scale studies and rather small sets of algorithms this can be done manually, but it is much harder for large performance studies that might take days to complete, where the hardness of the problem instances to be explored is inhomogeneous and difficult to predict.

A calibration algorithm should be equipped with another parameter, t_{opt}^{wct}, which is the wall clock time (WCT) the performance analyst desires as the duration of a single simulation run, averaged over all simulation algorithms and benchmark model instances. An optimal wall clock time should at least be in seconds or tens of seconds, to reduce the impact of operating system etc. Johnson also argues strongly against *"'the millisecond testbed"'* (Johnson 2003, p. 11), as he regards the performance of algorithms that solve problems that fast as irrelevant in most cases. However, this point does not hold here: the goal of such experiments is to *compare* algorithms that are usually executed for much longer. Finally, the user may also define a maximal wall clock time t^{wct}_{max}, which should not be exceeded by any execution of a benchmark input.

Since the benchmark model should have quasi-steady state characteristics within an interval of $[0, t^{sim}_{min}]$, its execution times for a larger interval of duration t_x can be approximated by extrapolation with the factor $x = t_x / t^{sim}_{min}$. This implies that the relative performance of the algorithms is not affected by letting them run longer: $\frac{x \cdot perf_a}{x \cdot perf_b} = \frac{perf_a}{perf_b}$. Hence, studies with calibrated simulation time intervals may need less time and still support the same conclusions than those studies without calibration, as long as the benchmark model fulfills the quasi steady-state property. However, a latter normalization of the results might be necessary to avoid biased analyses (Vuduc, Demmel, and Bilmes 2004, p. 81). The wall clock time saving of choosing t^{sim}_{min} instead of t^{sim}_{max} could be considerable, particularly if a large benchmark input space $\mathbb{B}$ has to be explored (see section 6).

4.1 A Simple Calibration Algorithm

A simple calibration algorithm may just use a sample set of algorithms $A \subseteq \mathbb{A}$ for searching a suitable simulation end time. Each algorithm could be applied once to the current benchmark model setup. If the algorithms are sufficiently representative for $\mathbb{A}$, averaging over their execution times should result in a rather good estimate of the overall average execution time. Sample size and the criterion for generating the sample are definable by the user (see section 6.2). The core of the very simple algorithm is briefly outlined in algorithm 1.

Algorithm 1 A simple algorithm for simulation end time calibration

```
1   public void calibrate(double[] durationsWCT) {
2           //Initialization [...]
3
4           //Calculate average WCT over all algorithms in the sample
5           $\overline{runtime_{wct}}$ = avg(durationsWCT);
6
7           //Check if current simulation end time is better then the best end time found so far
8           if(|bestAvgWCTime - $t^{wct}_{opt}| > |\overline{runtime_{wct}} - t^{wct}_{opt}|$) {
9                   bestEndTime = currentSimEndTime; bestAvgWCTime = $\overline{runtime_{wct}}$;}
10
11          //Check if calibration is finished since achieved WCT is close enough to optimum
12          if ($\overline{runtime_{wct}} \in [t^{wct}_{opt} \cdot (1-\theta), t^{wct}_{opt} \cdot (1+\theta)]$)
13                  done = true;
14
15          //Search by linear extrapolation; works because of quasi-steady state property of benchmark model
16          currentSimEndTime = max(min(currentSimEndTime * $t^{wct}_{opt}$ / $\overline{runtime_{wct}}$, $t^{sim}_{max}$),$t^{sim}_{min}$);}
```

The basic idea is that, since the benchmark model should have the quasi-steady state property for all simulation times $\geq t^{sim}_{min}$, a linear extrapolation can be used for searching a simulation end time with the desired characteristic (line 16). This is checked in line 12, where the average execution time of all algorithms in the sample is checked to be in the interval around t^{wct}_{opt}. The size of that interval can be controlled by another user-defined parameter $\theta \in [0,1]$. Finally, the user can also configure the maximal number of search steps, so that an overly long calibration phase can be avoided in principle. To underestimate t^{sim}_{end} is much less costly than to overestimate it (since execution is faster for smaller time intervals), so the algorithm is initialized with t^{sim}_{min} as the currently best simulation time. Line 16 ensures that the simulation end time candidates are always in $[t^{sim}_{min}, t^{sim}_{max}]$. If a simulator exhibits an execution time $t^{wct} > t^{wct}_{max}$ for a `currentSimEndTime` $> t^{sim}_{min}$, a watchdog procedure outside the main calibration algorithm stops testing the suitability of `currentSimEndTime` and re-sets the algorithm to try $\max(t^{sim}_{min}, \text{currentSimEndTime} \cdot t^{wct}_{max} / (1.05 \cdot t^{wct}))$ as a new candidate. Again relying on a simple linear interpolation, the new simulation end time should result in a wall clock execution time for this simulator that is just 5% below t^{wct}_{max} (and not smaller than t^{sim}_{min}). If t^{sim}_{min} is initialized correctly, i. e., not too small for the benchmark model at hand, and the steady-state property of the model holds, the algorithm ensures an end time t^{sim}_{end} that allows a valid performance analysis ($t^{sim}_{end} \geq t^{sim}_{min}$), while the average wall clock time performance of the algorithms should be near to t^{wct}_{opt}.

Enhancements The simple algorithm can be enhanced in many ways. For example, it should be straightforward to integrate simple prediction methods, e. g., nearest neighbor, that generate 'good guesses' for the initial t^{sim}_{end} by considering the solutions for rather similar setups of the benchmark model. When discrete-event simulators are to be evaluated, one could also alter the algorithm to calibrate the overall number of events to be simulated, instead of the simulation end time. While this would be a more 'natural' metric for such algorithms, it does not work for approximative methods (see section 6).

5 SIMULATION SPACE EXPLORATION

5.1 Requirements

Given that there is a good benchmark model to explore the performance of an algorithm set $\mathbb{A}$, it is still unclear how to automate the actual experimentation and what issues arise in practice. We identified three major kinds of experiments that are commonplace in performance studies on simulation algorithms, differentiated by their objectives:

- **Algorithm-centric (AC)**: Experiments that thoroughly explore the behavior of a single algorithm.
  Major concern: *Where are the bottlenecks of the algorithm? In which dimension does it scale? How does it compare to alternative approaches?*
- **Trade-Off-centric (TC)**: Experiments that identify regions in the benchmark parameter space where the order of algorithms with respect to a performance measurement, e. g., execution speed, is changing.
  Major concern: *Where does algorithm A start to outperform algorithm B?*
- **Exploration-centric (EC)**: Experiments that compare a set of algorithms.
  Major concern: *Which algorithm is best under which circumstances, i. e., benchmark model setups?*

Following the categorization of empirical research on algorithms from (Johnson 2003, p. 3), algorithm-centric studies are typically conducted to support *"'horse race papers"'*, i. e., publications of new algorithms, and shall evaluate the benefits of the new algorithm (in comparison to other implementations). In contrast, TC and EC studies are performed for *"'experimental analysis papers"'*, i. e., papers that provide comparative analyses for larger sets of algorithms. We distinguish between TC and EC studies based on their focus on either the specific trade-off points of algorithms, which might suffice for experimental analysis, or on a thorough algorithm performance exploration, even in regions where tipping points are unlikely. While EC studies are generally the most exhaustive ones and also contribute to the validation of the involved algorithms, TC experiments might be executed much faster by relying on extrapolation methods for finding 'interesting' regions of the overall performance space $\mathbb{P}$ – which could then be thoroughly investigated by EC experiments.

A major requirement for an architecture that automates runtime performance analysis of simulation algorithms is to support all three kinds of studies. All study types can be enhanced by sophisticated methods from statistics or experimental design. Making the integration of such methods as easy as possible will help simulationists in setting up effective and meaningful performance experiments. Finally, the architecture should allow for a calibration of the simulation end time as discussed in section 4, and should be able to exploit available resources to speed up the overall study.

5.2 A Flexible Performance Space Exploration Architecture

We developed a simple yet flexible architecture for automating simulator runtime performance evaluation in the context of JAMES II. Its major components are outlined in Figure 3. The central entity is `ISimSpaceExplorer`, an interface that extends `IExperimentSteerer`, a general interface for all components that are able to dynamically interfere with the execution of an experiment, e. g., algorithms for optimization or sensitivity analysis. The interface is implemented by the `AbstractSimSpaceExplorer` class, which is based on the Strategy pattern (Gamma et al. 1995, p. 315) for all explorations of $\mathbb{P}$. It handles the interplay between the (optional) calibration of the benchmark model's simulation end time by an implementation of `IModelCalibrator` on one hand, and the algorithm that chooses benchmark model setups, i. e., the elements from $\mathbb{B}$ to be explored, on the other hand. To do so, `AbstractSimSpaceExplorer` distinguishes three phases – `START_CALIBRATION`, `CALIBRATION`, and `EXPLORATION`:

1. The first phase is needed to initialize the calibrator and to select the benchmark model setup of interest, i. e., the element of $\mathbb{B}$ that shall be explored now. The element is selected by the concrete exploration algorithm, so the abstract method `newModelSetup()` is called.
2. In the `CALIBRATION` phase, the simulation explorer queries the calibrator and then propagates the tested end time t_{end}^{sim} and the sample algorithm to the experimental layer of JAMES II (Himmelspach, Ewald, and Uhrmacher 2008). It will continue to work as a proxy for the calibrator until `IModelCalibrator.done()` is true. Then, the best

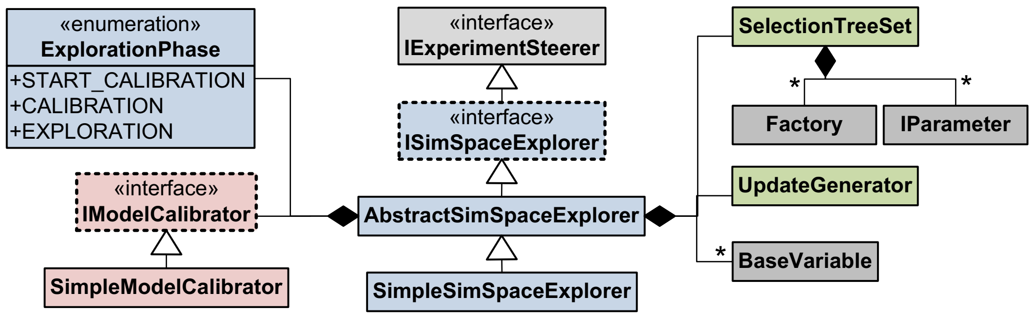

Figure 3: The main structure of the performance space exploration architecture: the simulation space explorer (blue), the calibrator (red), and the parameter block management (green). Most are linked with fundamental JAMES II entities (grey). Extension points, i. e., plug-in types, are marked by bold dotted borders.

found t_{end}^{sim} is retrieved from the calibrator and the `EXPLORATION` phase begins. If no calibrator is set, the default simulation end time will be used and this phase of the algorithm is skipped.

3. In the last phase, the explorer selects the elements of $\mathbb{A}$, i. e., the algorithms to be tested on the current benchmark model setup. This is done by continuously calling the abstract method `explore()` until it returns `null`. The abstract `nextProblem()` method is called to decide whether the exploration has finished or the whole process starts over again.

Plug-in types & auxiliary components Both calibration and exploration algorithms, i. e., implementations of `IModelCalibrator` and `ISimSpaceExplorer`, are likely to incorporate methods from experimental design or additional heuristics. We therefore decided to define *plug-in types* for them, i. e., extension points for the plug-in system of JAMES II (Himmelspach and Uhrmacher 2007b). This means that both calibrators and exploration algorithms can be exchanged quite easily. Furthermore, the plug-in system makes it possible to automatically retrieve all combinations of algorithms that are able to simulate a given benchmark model. Since simulation algorithms in JAMES II are usually not built in a monolithic manner, but rather by combining existing plug-ins in new ways, each of these combinations is treated as a single simulation algorithm: the performance of a discrete-event simulator, for example, might be greatly affected by the event queue plug-in it is combined with (Himmelspach and Uhrmacher 2007a). Configuring JAMES II to use a certain algorithm, i. e., a particular combination of plug-ins, requires to define a `ParameterBlock`. This is a generic tree structure that holds the parameters for the algorithms involved. Generating corresponding `ParameterBlock` instances for all suitable algorithm combinations is done by a `SelectionTreeSet` instance, which does so by analyzing entities from the plug-in system (`Factory`, `IParameter`). The model's parameter space is defined by a set of `BaseVariable` instances, which belong to the experimentation system. Finally, the `UpdateGenerator` implements the automatic generation of updates for key simulation parameters. It is required for the automatic calibration of the simulation end time.

Adaptive simulation runner JAMES II delegates the management of simulation run executions to a *simulation runner* component, which is also implemented as an exchangeable plug-in. In case of trade-off- or algorithm-centric experiments, one could harness the *adaptive simulation runner* (Ewald, Leye, and Uhrmacher 2009) to reduce the computational effort when testing a single parameterization of the benchmark model at hand. This is done by exploiting policies for the multi-armed bandit problem, i. e., policies that automatically identify the best algorithm by subsequently choosing an implementation and interpreting their runtime performance on the fly (see section 2 or (Auer, Cesa-Bianchi, and Fischer 2002)). The better an algorithm performs, the more often will it be chosen for execution. This allows us to reduce the number of required replications. For example, when investigating a set of 30 algorithms, instead of replicating the execution of each algorithm 20 times ($30 \cdot 20 = 600$), the adaptive simulation runner can carry out the choice of interesting algorithms on its own and just execute 300 replications. In this example, we could reduce the number of required replications by 50% and will still have *more* than 20 replications, i. e., *more* experimental data, for all algorithms that perform relatively well (on the expense of the worse-performing algorithms).

Performance data storage and analysis The experimental data that is generated by the performance exploration system needs to be stored for later analysis. This is done by a performance recorder that feeds the data directly into the JAMES II performance database (Ewald, Himmelspach, and Uhrmacher 2008). Afterwards, the data can be analyzed with

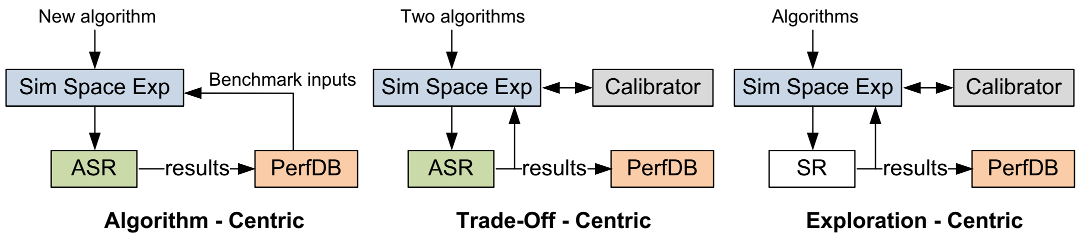

Figure 4: Different setups of the automatic performance exploration architecture

a custom data mining framework, SPDM (Ewald, Uhrmacher, and Saha 2009), which provides wrappers for several popular data mining systems (e. g., WEKA (Witten and Frank 1999)).

Configuring the architecture for AC/TC/EC studies Figure 4 shows how to set up the performance space exploration architecture for the different kinds of studies described in section 5.1. When evaluating the performance of a new algorithm (left schema), an algorithm-centric study can simply re-use suitable benchmark model setups from the database. By taking advantage of the adaptive simulation runner, the number of required replications can be greatly reduced by letting it decide between the new algorithm and the fastest known algorithm for this problem instance, identified by the performance database. Having only two options will make a multi-armed bandit policy converge quickly.

If only the trade-off points between two algorithms shall be found (center schema), the exploration algorithm should take the algorithm performance directly into account, so that those regions can be identified in which tipping points are likely (by specialized implementations of `ISimSpaceExplorer`). Since these regions are usually unknown before, the calibration mechanism should be applied to newly created benchmark model setups. Again, the adaptive simulation runner can be used to reduce the number of required replications. A thorough exploration of all available algorithms (right schema) also includes to explore the variance of each algorithm throughout the benchmark model parameter space. Therefore, using the adaptive simulation runner is not advisable for this setup, since it may execute badly performing algorithms only once per problem instance. Similar to the setup for trade-off - centric studies, obtained results are fed back to the exploration algorithm, so that it can decide upon new problem instances to be investigated. Exploration algorithms for TC and EC studies could be inspired by similar problems in optimization, artificial intelligence, or statistical modeling.

6 EVALUATION

6.1 Benchmark Models

To illustrate the functioning of our architecture, we applied it to investigate the performance of stochastic simulation algorithms (SSA), a family of discrete-event algorithms that can be used to simulate biochemical reaction networks in Computational Biology (Gillespie 1977). For illustrating the use of quasi-steady state benchmarks models, we apply τ-leaping (Cao, Gillespie, and Petzold 2006), an approximative SSA variant, to two benchmark models: the *Linear Chain System* (LCS (Cao, Li, and Petzold 2004)) and the *Cyclic Chain System* (CCS (Jeschke and Ewald 2008)), both of which have been developed for benchmarking SSA performance. LCS represents a linear chain of reactions between species X_i, having the form $X_1 \rightarrow X_2$, $X_2 \rightarrow X_3$, $\ldots X_{n-1} \rightarrow X_n$. We initialized it with 100 reactions and therefore 101 species. In its initial state, there are only particles of the species that is first in the chain, i. e., X_1. We set $|X_1| = 10^7$. It is clear from the structure of LCS that there is no steady state it could ever be in, apart from the end state where $|X_1| = \ldots = |X_{n-1}| = 0$ and $|X_n| = 10^7$.

CCS, in turn, was designed to stay in a steady state. Although the simulated reactions change the state of the system, there is always a reverse reaction whose propensity will then increase. It is called a cyclic system because all reactions are of the form $X_i + X_{i+1} + \ldots \rightarrow X_{i+j} + \ldots$, where all indices are calculated modulo the number of available species. The CCS has three parameters that are worth exploring: number of species, number of reactants per reaction, and number of reactions per species. For the first experiment, they have been set to ten, three, and one, respectively. The number of reactions per species also equals the number of products per species. In the initial state of CCS, each species has the same amount of particles. We set $|X_i| = 10^5$ in all experiments. Since CCS is in a steady sate, the amounts of the X_i only vary slightly over time. The results in Figure 5 suggest that CCS can be calibrated (see section 4.1)). LCS simulation speed, on the other hand, grows nonlinearly for τ-leaping (we used the default parameters from (Cao, Gillespie, and Petzold 2006)). This does *not* hint at any methodological problem in previous studies, as these did not rely on simulation time interval calibration. It just shows that there are benchmark models, like LCS, which are unsuitable for this calibration mechanism.

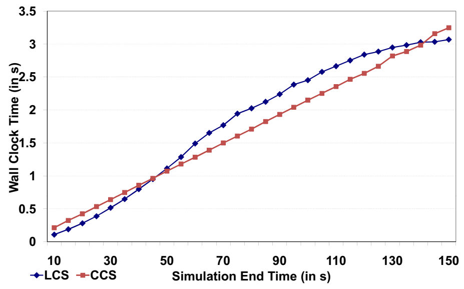

Figure 5: Steady-state behavior of CCS, in contrast to LCS. Averaged over 10 replications per setup.

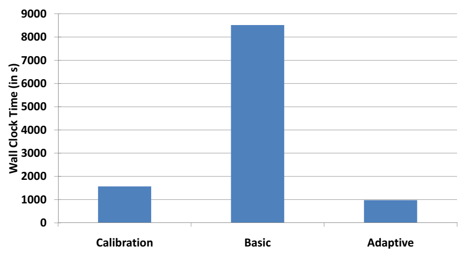

Figure 6: Overall CPU time of setups. Calibration has a strong impact on overall experiment performance.

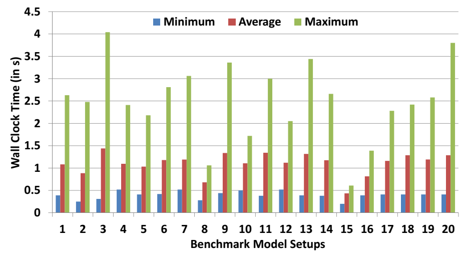

Figure 8: Minimum, average, and maximum wall clock time for algorithms applied to the calibrated setups from Figure 7

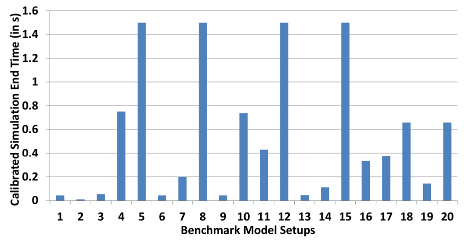

Figure 7: Calibrated t^{sim}_{end} times per benchmark model setup

6.2 Exploration Schemes

We evaluated the functioning of the architecture by letting different setups explore the same elements of the CCS parameter space $\mathbb{B}$: each instance had between 2 and 10 species, between 1 and 5 reactions per species, and between 1 and 10 reactants per reactions. Hence, $|\mathbb{B}| = 9 \cdot 5 \cdot 10 = 450$, from which we randomly sampled 20 benchmark model setups. As elements of $\mathbb{A}$, we chose 14 algorithms: one is the Direct Method (DM, (Gillespie 1977)) and the others are variants of the Next Reaction Method (NRM, (Gibson and Bruck 2000)). The latter can be combined with one of the various event queue implementations from JAMES II (Himmelspach and Uhrmacher 2007a), from which we selected 13. A similar study has already been presented in (Jeschke and Ewald 2008), but here we focus on *automating* performance experiments, and therefore test the novel calibration algorithm and its interplay with the adaptive simulation runner. The calibrator parameters were $t^{wct}_{opt} = 1s$, $t^{wct}_{max} = 20s$, $t^{sim}_{min} = 0.01s$, $t^{sim}_{max} = 1.5s$, and the tolerance $\theta = 30\%$. A sample of three algorithms was considered. Configurations with differing simulation schemes were preferred over those that just varied in their event queue implementation – in this simple case, the calibrator always sampled the DM simulator along with two combinations of NRM and event queue. Apart from the setup with the adaptive simulation runner, where the number of replications was set to $2 \cdot |\mathbb{A}| = 28$, every algorithm in $\mathbb{A}$ had to be replicated four times for each of the 20 benchmark inputs, resulting in $14 \cdot 4 \cdot 20 = 1120$ simulation executions. We tested three setups: `Calibration` (calibration activated), `Basic` (no calibration, $t^{sim}_{end} = 0.545709$, which is a rather good guess that equals the overall average calibrated t^{sim}_{end} times from `Calibration`), and `Adaptive` (calibration and adaptive simulation runner activated).

Figure 6 shows the overall results in terms of the CPU time it took each setup to explore the desired fraction of the performance space, i. e., to apply the 14 algorithms to the 20 benchmark model setups. The exploration of unknown model setups is much faster when using `Calibration` instead of `Basic`, with a speed-up of more than 500%. `Adaptive`, in turn, outperforms `Calibration` with a speed-up of 60%, but this is only due to the reduced number of replications (on top of calibration). In Figure 7, the simulation end times found by `Calibration` are plotted, and Figure 8 depicts the corresponding wall clock time statistics. It can be seen that the calibration works as intended: while the calibrated t^{sim}_{end} times vary greatly (Figure 7), the average wall clock times are seldom outside their tolerance region of +/- θ, i. e., +/- 30%. All experiments in this section have been conducted on a 2.0 GHz two-core AMD 3800+ CPU with 1GB RAM and a Java SciMark (Scimark) score of 230.2 Mflops.

7 CONCLUSIONS

In this paper, we discussed how experiments concerned with the runtime performance of simulation algorithms can be automated. Firstly, we argued for the usage of synthetic benchmark models that are parameterizable, scalable, simple, and in a quasi-steady state (section 3). We then introduced a simple yet effective algorithm to calibrate the simulation end time for such benchmark models (section 4). Finally, we presented a simple architecture for the realization of various experimentation strategies. It can be easily reconfigured to serve different objectives: algorithm-centric, trade-off-centric, or exploration-centric (section 5). The functioning of the presented architecture has been shown by applying it to a realistic research problem (section 6). Although the algorithms that were used to improve experimentation are quite simple – and hence easy to implement in other systems as well – we could reduce the execution time of a rather typical performance study by more than 850%.

We hope that our results will stir future research in this area, as there are still several open issues. For example, some algorithms like τ-leaping have large parameter spaces on their own. The parameter space of the τ-leaping algorithm described in (Cao, Gillespie, and Petzold 2006) is four-dimensional, and the parameters affect both accuracy and execution speed. Such algorithms are still very challenging to analyze – the presented solutions may work for rather large sets of algorithms, but not for *really* large sets, where additional methods for interpolation and statistical analysis are required, such as (Ankenman, Nelson, and Staum 2008). These interpolation methods might be able to efficiently identify interesting regions of such parameter spaces. In the end, the presented methods are intended to automate research on simulation algorithms as much as possible, which is highly desirable in all kinds of experimentation-guided research (Waltz and Buchanan 2009). It should make the life of simulationists easier and might also contribute to the quality of simulator algorithms, as they can now be tested *both* more easily and more thoroughly. In the near future, we plan to consider more sophisticated criteria for stopping a performance study, and to integrate various experiment design methods. Current releases of JAMES II can be found at `http://www.jamesii.org`.

ACKNOWLEDGMENTS

This work has been supported by the German research foundation (DFG).

REFERENCES

Agrawal, N., A. C. Arpaci Dusseau, and R. H. Arpaci Dusseau. 2008. Towards realistic file-system benchmarks with CodeMRI. *SIGMETRICS Perform. Eval. Rev.* 36 (2): 52–57.

Ankenman, B., B. L. Nelson, and J. Staum. 2008. Stochastic kriging for simulation metamodeling. In *Proceedings of the 2008 Winter Simulation Conference*, ed. S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, and J. W. Fowler, 362–370. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Auer, P., N. Cesa-Bianchi, and P. Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.* 47 (2-3): 235–256.

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.

Box, J. F. 1980. R. a. fisher and the design of experiments, 1922-1926. *The American Statistician* 34 (1): 1–7.

Cao, Y., D. T. Gillespie, and L. R. Petzold. 2006, Jan. Efficient step size selection for the tau-leaping simulation method. *J Chem Phys* 124 (4).

Cao, Y., H. Li, and L. Petzold. 2004. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *J Chem Phys* 121 (9): 4059–4067.

Cheeseman, P., B. Kanefsky, and W. M. Taylor. 1991. Where the really hard problems are. In *Proceedings of 12th International Joint Conference on AI (IJCAI-91)*, ed. J. Mylopoulos and R. Reiter, Volume 1, 331–337.

Ewald, R., J. Himmelspach, and A. M. Uhrmacher. 2008. An algorithm selection approach for simulation systems. In *Proc. of the Workshop on Principles of Advanced and Distributed Simulation 2008*, Volume 22, 91–98: IEEE Computer Society.

Ewald, R., S. Leye, and A. M. Uhrmacher. 2009. An efficient and adaptive mechanism for parallel simulation replication. In *Proc. of the Workshop on Principles of Advanced and Distributed Simulation 2009*, 104–113: IEEE CPS.

Ewald, R., A. Uhrmacher, and K. Saha. 2009. Data mining for simulation algorithm selection. In *Proceedings of the SIMUTools'09: 2nd International Conference on Simulation Tools and Techniques*.

Fellows, M. R. 2001. Parameterized complexity: The main ideas and some research frontiers. In *Algorithms and Computation*, ed. P. Eades and T. Takaoka, Volume 2223 of *Lecture Notes in Computer Science*, 291–307: Springer.

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

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

Gagliolo, M., and J. Schmidhuber. 2006, August. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence* 47 (3-4): 295–328.

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

Gibson, M. A., and J. Bruck. 2000. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *J. Chem. Physics* 104:1876–1889.

Gillespie, D. T. 1977. Exact Stochastic Simulation of Coupled Chemical Reactions. *Journal of Physical Chemistry* 81 (25).

Gittins, J. C. 1989, January. *Multi-armed bandit allocation indices (wiley interscience series in systems and optimization)*. John Wiley and Sons Ltd.

Glinsky, E., and G. Wainer. 2005. Devstone: a benchmarking technique for studying performance of devs modeling and simulation environments. In *Proc. of the Distributed Simulation and Real-Time Applications Symposium 2005*, 265–272.

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. Mönch, O. Rose, T. Jefferson, and J. W. Fowler. 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. Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium*, 137–143: IEEE Computer Society.

Hogg, T., B. A. Huberman, and C. P. Williams. 1996, March. Phase transitions and the search problem. *Artificial Intelligence* 81 (1-2): 1–15.

Jeschke, M., and R. Ewald. 2008. Large-scale design space exploration of ssa. In *Computational Methods in Systems Biology, International Conference, CMSB 2008, Rostock, Germany, October 12-15, 2008, Proceedings*.

Johnson, D. 2003, February. *Data structures, near neighbor searches, and methodology*, Chapter A theoretician's guide to the experimental analysis of algorithms, 215–250. Oxford, United States: American Mathematical Society.

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

Jun, T. 2004, December. A survey on the bandit problem with switching costs. *De Economist* 152 (4): 513–541.

LaMarca, A., and R. E. Ladner. 1997. The influence of caches on the performance of sorting. In *SODA '97: Proc. of the 8th annual ACM-SIAM symposium on discrete algorithms*, 370–379. Philadelphia, USA: Soc. for Industrial and Applied Mathematics.

Low, M. Y. H. 2002. Dynamic load-balancing for bsp time warp. In *Simulation Symposium, 2002. Proceedings. 35th Annual*, 267–274.

McGeoch, C. 1992. Analyzing algorithms by simulation: variance reduction techniques and simulation speedups. *ACM Comput. Surv.* 24 (2): 195–212.

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

Plackett, R. L., and J. P. Burman. 1946. The design of optimum multifactorial experiments. *Biometrika* 33 (4): 305–325.

Ramakrishnan, N., J. R. Rice, and E. N. Houstis. 2002, January. Gauss: an online algorithm selection system for numerical quadrature. 27–36.

Rice, J. R. 1976. The algorithm selection problem. *Advances in Computers* 15:65–118.

Robbins, H. 1952. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society* 58 (5): 527–535.

Scimark, J. http://math.nist.gov/scimark2/.

Skadron, K., M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. 2003, August. Challenges in computer architecture evaluation. *Computer* 36 (8): 30–36.

Small, C., N. Ghosh, H. Saleeb, M. Seltzer, and K. Smith. 1997, November. Does systems research measure up? Technical Report TR-16-97, Harvard University.

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. Mönch, O. Rose, T. Jefferson, and J. W. Fowler, 689–694. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Vermorel, J., and M. Mohri. 2005. Multi-armed bandit algorithms and empirical evaluation. 437–448.

Vuduc, R., J. Demmel, and J. Bilmes. 2001. Statistical models for automatic performance tuning. In *Computational Science ICCS 2001*, 117–126. Springer Berlin / Heidelberg.

Vuduc, R., J. W. Demmel, and J. A. Bilmes. 2004, February. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.* 18 (1): 65–94.

Waltz, D., and B. G. Buchanan. 2009, April. Computer science: Automating science. *Science* 324 (5923): 43–44.

Witten, I. H., and E. Frank. 1999, October. *Data mining: Practical machine learning tools and techniques with java implementations*. Morgan Kaufmann.

Yu, H., D. Zhang, and L. Rauchwerger. 2004. An adaptive algorithm selection framework. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, 278–289.

AUTHOR BIOGRAPHIES

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 email address is <roland.ewald@uni-rostock.de>.

ADELINDE M. UHRMACHER is an Associate Professor at the Department of Computer Science at the University of Rostock and head of the Modeling and Simulation Group. Her research interests are in modeling and simulation methodologies and their applications. Her e-mail address is <lin@informatik.uni-rostock.de> and her Web page is <www.informatik.uni-rostock.de/~lin>.