A Simulation Approach to Facilitate Parallel and Distributed Discrete–Event Simulator Development

Roland Ewald, Jan Himmelspach, Adelinde M. Uhrmacher
Institut für Informatik
University of Rostock, Germany
{re027, jh194, lin}@informatik.uni-rostock.de

Dan Chen, Georgios K. Theodoropoulos
School of Computer Science
University of Birmingham, UK
{D.Chen, gkt}@cs.bham.ac.uk

Abstract

Efficiently simulating discrete-event models in a parallel and distributed manner is a challenging endeavour. On one hand, various factors, such as hardware infrastructure or model characteristics, have to be considered. On the other hand, there is a wide variety of algorithms which address subproblems of parallel and distributed simulation and whose performance depends on the application at hand. We illustrate the resulting difficulties with respect to the development of parallel and distributed simulation systems and argue that the simulation of distributed simulation systems is a feasible approach to alleviate them. To underpin this, we introduce SIMSIM, *a sequential simulator for parallel and distributed simulation systems.* SIMSIM*'s pertinency is illustrated by the development of a load balancing algorithm for PDEVS. The algorithm's performance is analysed using* SIMSIM *and the predicted performance is compared to the performance of its implementation in the simulation system* JAMES II.

1. Introduction

Developers of distributed simulation systems face various problems: Besides specific algorithms to address problems of distributed simulation systems (e.g., synchronisation, partitioning, or load balancing) [9], other fundamental problems of distributed systems – like error recovery, interfaces to external software systems, or side-effects of network protocols – may have to be considered as well.

On top of this, research on parallel and distributed discrete-event simulation (PDES) is often focused on a certain application at hand, e.g. computer networks [20] or particle movement [21]. While this allows to increase the efficiency of PDES for the addressed subset of problems, it also hampers the comparability and repeatability of PDES approaches in general. Moreover, the PDES algorithms to be compared are implemented in different programming languages and usually tailored to a specific infrastructure, i.e. to a specific type of processor (see [4]), a specific communication layer (e.g., MPI or RMI), or other characteristics of the environment.

These circumstances make it very challenging to select efficient algorithms for a general-purpose PDES system, or to develop new ones. For example, the performance of a newly developed load balancing algorithm could depend on the initial partitioning algorithm, the algorithm to route the information between different logical processes (LPs), or even the way the simulation is synchronised. Any performance observation of such an algorithm is therefore biased.

Of course, it is possible to analyse an algorithm's performance by testing it in various setups, so that the influence of other mechanisms, as well as its sensitivity to certain parameters (e.g., network latency) can be estimated. However, two aspects prohibit this approach for most cases: *Firstly*, it is tedious and expensive to test an algorithm within a running PDES system. Due to the large number of possibly interfering aspects (e.g., parameter settings), a large number of tests has to be conducted for sensitivity investigations. The influence of external factors like background processes has to be minimised as well. Furthermore, the unbiased observation of the execution is hard to achieve [15]. *Secondly*, the generalisation of test results is complicated. The implementation quality of the algorithm and the PDES system it is running in is often unclear, and effects of programming language, operating system, or model properties are hard to quantify. This makes the comparison of different algorithms very hard, or even impossible [17].

To circumvent the second problem, one could implement all algorithms in a common environment. This was in fact one motivation for the development of the simulation system JAMES II [13]. To avoid time-consuming testing, however, one would have to *predict* the performance of a

PDES system under given circumstances. This paper describes the simulation of a PDES system execution – so to say, a meta-simulation – as a performance prediction approach that addresses the aforementioned problems.

2. Background and Related Work

In recent years, various approaches to predict and analyse the performance of PDES algorithms have been developed. These approaches can be roughly categorised into analytical approaches that employ mathematical methods, empirical approaches that use empirical data from experiments, and mixed approaches.

Mathematical models are very useful to obtain general knowledge of PDES algorithms, since they abstract most of the influencing parameters away. This allows the identification of theoretical upper and lower performance bounds [19]. On the other hand, mathematical models are hardly tractable unless several very strong assumptions are made, e.g. negligible roll-back cost, zero-delay communication, or a homogeneous set of processors [12]. Besides that, the formal description of an algorithm's behaviour might be a very challenging task in itself. Nicol concluded that *"[...] synchronization behaviour is frequently complicated, which makes it very difficult to analytically prove anything about performance executing large models on large machines."* [19, p. 4].

Mixed approaches integrate empirical data with mathematical models, so that aspects which are difficult to model mathematically can be replaced by real-world measurements (e.g., communication delay, etc.). Liu et al. used benchmark results as parameters for a simple mathematical model [15]. Although their predictions were quite precise, they reported that it was very difficult to obtain accurate empirical data, and that the whole process has to be repeated manually when the performance of another PDES system ought to be predicted. Juhasz et al. developed a trace-based approach to predict simulation performance [14]. They analysed a trace of a sequential simulation by transforming it into a dependency graph of the processed events. A critical path analysis was used to calculate the possible speed-up for using PDES instead. By altering the graph, specific hardware setups could be taken into account and the impact of different algorithms could be estimated (see [22]). Although this approach is very promising, the underlying graph manipulation algorithms are often rather costly, so that the scalability of this approach is unclear [15]. Another problem of trace-based analysis is the validation of (possibly complex) algorithms for re-labelling or re-structuring the dependency graph, since errors will only lead to wrong results instead of program errors, and are therefore hard to identify.

Empirical approaches observe real PDES executions to predict performance. Perumalla et al. virtualised an existing PDES system by simulating its distributed execution on a smaller set of processors [21]. Their approach aimed at saving debugging and testing time on costly supercomputer infrastructure. In contrast, the simulation approach presented here focuses on facilitating the *development* of PDES algorithms and systems. While this also includes support for testing and debugging, these activities take place on a more abstract level, and for different purposes. Our approach aims at assisting the developer in finding design problems of an algorithm *prior* to its actual implementation, and to test it by predicting its performance under a large number of circumstances. There are other empirical approaches for the simulation of distributed systems in general (e. g., [2]), but these do not address the specific needs of PDES system development (see 3.1).

3. SimSim

The terminology used in the domain of PDES systems may lead to confusion when describing a simulation of the very same. At first, we distinguish between *system model* and *application model*. The system model is the model of the PDES system, thus it is simulated by SimSim. Since the system model in turn represents a simulation system, this system simulates a model as well, which is called the application model. In [21], additional timelines for PDES system simulation have been identified. To avoid misunderstanding, we will explicitly name the timelines that correspond to system and application model: The *simulated wall-clock time* (SWT) is the simulation time of SimSim, i.e. the system model's notion of time. The time with respect to the application model is called *application model time* (AMT).

3.1. Requirements

To support different PDES systems, a simulation tool has to be extensible and flexible, especially with respect to observable measurements and the nature of algorithms under investigation (e.g., granularity control, load balancing, etc.). At the same time, such a simulation tool should support the developer by providing as much predefined functionality as possible. An expedient combination of both requirements would be to predefine the underlying structure of PDES systems in general, while providing a clear interface to extend it. In [18], Misra introduced the notion of *logical processes*, which are assigned to *physical processors* and exchange *messages*. This concept is widely accepted and general enough to include all kinds of PDES systems, so it was chosen as a *generic PDES system model* for SimSim. The developer of a PDES system simulation can rely on this generic model, and only has to account for the characteristics of the specific PDES system at hand, i.e., for the *specific PDES system model.*

Another requirement arises when considering different usage scenarios. Besides the performance prediction for various setups, which allows the identification of an algorithm's flaws and its sensitivity regarding certain parameters, it might also be important to scrutinise a single simulation run. This allows to investigate the algorithm's behaviour in special cases and to find the causes of incorrect behaviour. Since debugging of real PDES systems is often difficult and possibly expensive [21], a PDES system simulator should provide means to help the developer in the aforementioned cases, albeit on an abstract level (i.e., only flaws in SIMSIM's model of the PDES system can be detected, not flaws in the PDES system itself). The distinction between executing a batch of simulation experiments and a single experiment leads to additional needs. While the simulator should run as fast as possible when in batch mode, debugging requires adjustment of the simulation speed. Moreover, visualisation of the PDES system model is helpful to conceive the current status of the simulated PDES system immediately.

3.2. Features

A sequential PDES system simulator, SIMSIM, was developed to illustrate the benefits of meta-simulation during PDES system development. SIMSIM was designed to fulfil the requirements outlined in the last section. Firstly, a plug-in interface was realised, which allows the developer to define specific PDES system models. Until now, SIMSIM does not support any modelling formalism to facilitate the definition of PDES system models. This is partly because any software modelling formalism could be used (e.g., state charts, flow charts, etc.), so that a selection is difficult and requires further research. Additionally, there are some advantages in allowing the extension of the software by program code. The developer does not have to learn a new modelling formalism, but is able to use the common constructs of imperative programming languages. This makes the model of the PDES system a working prototype that lacks the communication layer (emulated by SIMSIM) and anything else the modeller is not interested in. Such a prototype is a good starting point for a real implementation. Nevertheless, a concise modelling formalism to describe PDES systems could facilitate the use of SIMSIM and should be subject to future research.

Secondly, SIMSIM distinguishes between the two simulation modes outlined in section 3.1, namely the *batch mode* (i.e., batch job execution as fast as possible) and the *interactive mode* (i.e., a single, paced simulation the user can interact with). A graphical user interface allows the visualisation of interactive simulation runs and facilitates the definition of experimental setups, which can be stored in XML files. To support the setting of PDES system specific parameters, the user interface is also extensible via the plug-in mechanism. Finally, SIMSIM allows to integrate external simulators for special purposes. For example, an external network simulator might be useful to investigate specific network protocols or topologies and how they influence the performance of a PDES system.

3.3. Realisation

3.3.1 Architecture

SIMSIM is implemented in Java and can be seen as a generalized, revised, and enhanced version of the PDES-MAS simulator [7], which was tailored to predict the performance of routing algorithms in the PDES system PDES-MAS [16]. The old simulator was transformed into a PDES-MAS plug-in for SIMSIM. In addition, a JAMES II plug-in was developed, which models the abstract PDEVS simulator of JAMES II. Since both PDES system models are quite different – e.g., regarding modelling entities (different types of LPs vs. DEVS coordinator and simulator), synchronisation protocol (optimistic vs. conservative), and purpose (multi-agent systems vs. DEVS models)–, SIMSIM is shown to be generic enough to subsume various kinds of PDES systems.

However, the price for this generality is a rather abstract model of physical processors and logical processes. Therefore, it might be desirable to not only allow different specific PDES system models, which are encapsulated by plug-ins, but also different generic PDES system models. This was realised by employing the strategy pattern [10, p. 315 - 323] for most of the core classes. If more sophisticated models of hardware or LPs are needed, one can simply exchange the corresponding components.

3.3.2 Execution

The basic simulation approach of SIMSIM is quite simple: A set of LPs is executed on a set of modelled processors, which in turn are connected via a modelled network. Each message that is sent from one LP to the other is handled as an event in SIMSIM, and therefore added to a central event queue. All events are timestamped with the time at which their corresponding message will reach its destination.

After the simulation started and initial messages have been added to the event queue, a single simulation loop is executed: The event with the smallest time stamp is removed and handed over to the physical processor on which the destination LP is situated. The physical processor registers the event execution of the LP and passes the message on. The LP itself serves as a wrapper for the specific PDES system model, which was provided by the plug-in developer. Now, this code can react to the message by generating

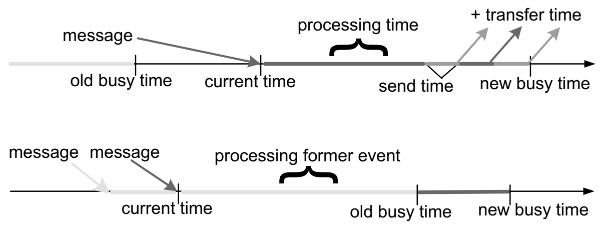

Figure 1. Send time calculation

new messages and 'send' them to other LPs, which means that the arrival of each generated message is scheduled as a new event. After the code to handle the message is executed, the next event with the smallest time stamp is chosen, and the whole procedure continues. The loop stops when the event queue is empty or the user interrupts it.

While the general scheme is straightforward, the actual calculation of message arrival times is not trivial. For each received or sent message, the plug-in code has to define corresponding *processing* times[1]. Since processing times depend on the speed of the host processor, given times can only relate to a (possibly hypothetical) baseline system. To simulate the execution in a heterogeneous environment, each processor is initialised with a computation power factor cp, which defines its performance with respect to the baseline system. This factor is then multiplied by the processing times, e.g. $cp = 0.5$ defines a processor to be twice as fast as the baseline system. It is also important to notice that the definition of processing time does not need to have any relation to its *wall clock* processing time in SIMSIM. In fact, these times can be chosen at will, so that they are useful for the plug-in developer to precisely control the granularity of the application model (i.e., the ratio of computational effort per event to communication overhead).

The size of a message has to be specified by the plug-in code as well, because this may have an impact on its transfer time in the network. To compute the transfer time of a message between two processors, the model of Juhasz et al. [14] was modified[2]: $T_{transfer} = T_s + k \cdot T_w$.

T_s is the time to initialise a message of k bytes, and T_w is the transfer time overhead per byte. In SIMSIM, the parameters T_s and T_w can be set for *each* connection between two processors. Therefore, it is possible to simulate a PDES system's performance under very different circumstances, from efficiently connected clusters to sparsely connected high-latency networks. Putting these things together, a processor can now determine an LP's processing time until it sends a message, whose arrival time can then be calculated by simply adding the transfer time. However, it has to be made clear that this approach implies a single-threaded execution, i.e. each physical processor executes the LPs one after another, corresponding to the order of messages it received (see figure 1). Besides resulting in slightly different event time stamps, this also means that no multi-threading overhead is considered.

[1]These times must not be confused with the time the network needs to transfer a message.

[2]Sending messages between unconnected processors has to be implemented manually, Juhasz et al. used an additional parameter to account for this.

4. PDEVS

To show the feasibility of using SIMSIM in the development process of a PDES algorithm, we used it to develop a load balancing algorithm for the abstract PDEVS simulator of JAMES II. PDEVS is a DEVS [23] based modelling formalism with inherent support for parallelism. As for standard DEVS, the execution semantics of PDEVS is specified by an abstract simulator. PDEVS models are either atomic models, which are computed using so-called *simulators*, or coupled models, which are computed by so-called *coordinators* and consist of other atomic or coupled PDEVS models. So, coupled models are used for nesting, which results in a hierarchy of atomic and coupled PDEVS models that is called model tree. The corresponding hierarchy of coordinators and simulators is called abstract simulator tree. Often, both terms are used synonymously, since each model entity is handled by exactly one simulation component.

The simulation is controlled by the so-called root coordinator, which is the parent of the topmost coordinator. To initiate a simulation step, the root coordinator creates a *-message which is then propagated to all atomic models that ought to execute an internal state transition at the current point in simulation time. The receipt of a *-message by an atomic model leads to the generation of output, which is then forwarded to other atomic models. Finally, all models that received output or a *-message execute a state transition and notify their parent via a *done*-message upon completion. When the root coordinator receives a *done*-message, it can proceed with the next simulation step.

All in all, there are quite many messages to be processed for a single simulation step. All simulators of models which have to compute an *internal* state transition (i.e., they received a *-message) need to exchange four messages with their parent. Those who only have to execute an *external* state transition (i.e., they did not receive a *-message, but output generated by other atomic models) still need to exchange two messages. The same holds for all coordinators of coupled models that are involved.

5. A Load Balancing Algorithm for PDEVS

In this context, load balancing means the re-partitioning and re-distribution of the simulated model at runtime. Load balancing makes sure that no processor is overly slowed down by too much load, so that the collapse of single processors can be avoided and parallel execution may be optimised. The question is now, under which circumstances the application of a certain load balancing algorithm is beneficial (and to what extent). This is where SIMSIM comes into play, besides its aforementioned use for debugging and prototyping. In addition to hardware, network, and PDES system parameters, a load balancing algorithm may also rely heavily on model characteristics, e.g. the degree of dynamism it exhibits, so that a simulation-supported development of this kind of algorithm seems favourable. Since load balancing is a fundamental problem of many distributed systems, various approaches have been developed, investigated, and compared (e.g., in [24]).

Load balancing algorithms for PDES systems strive to enhance their performance by taking simulation-specific information into account, so that there are many different approaches which are tailored to a certain protocol or application (e.g., [3]). Likewise, the algorithm presented here is built around the specific requirements of the abstract PDEVS simulator in JAMES II. To test its sensitivity to certain model parameters, the next section will outline a benchmark model. Afterwards, different crucial parts of the algorithm are detailed.

5.1. A Benchmark Model

To test the load balancing algorithm, the JAMES II plug-in was extended by a benchmark application model. Different benchmark solutions for PDES systems in general [1] and specifically for DEVS models [11] have been proposed, but these solutions do not account for model dynamism, i.e. model entities that change their behaviour over time.

The benchmark model we implemented resembles the PHOLD model [8] in that the atomic PDEVS models are exchanging messages among each other. To have control over model-inherent clusters, all atomic models are organised in groups. For each model, a preference factor $pref$ determines to which extent it exclusively exchanges messages with group members. If $pref = 1$ for all models, they only communicate within their groups, while $pref = 0$ makes grouping irrelevant. A model creates a new message after an amount of simulation time , which follows a random distribution, has gone by. Similarly, a received message will be delayed for a certain simulation time, before it will be returned to the sender, which will then destroy it. When a model receives or sends a message, random distributions determine how much processing time (i.e., simulated wall-clock time) it needs to do so.

All parameters for the aforementioned random variables define a single *state phase* of a model. A state phase has a certain duration in application model time. When a state phase expires, the corresponding model may change its group with a certain probability, P_{change}. Then, the next state phase of the model is selected to define the model's behaviour. Each model can have an arbitrarily long list of state phases. Therefore, it is possible to not only define active and passive model entities, but also entities which change their behaviour more or less rapidly and often.

5.2. Basic Algorithm

The abstract PDEVS simulator uses parallel execution only for events that occur simultaneously. To record parallel executions of its sub-components, each coordinator maintains a *parallelism matrix* P, which stores the number of pairwise parallel executions of its sub-components i and j at $P_{i,j}$. The overall number of a sub-component i's executions (i.e., how many times it performed a state transition) is recorded in $P_{i,i}$. If a larger number of sub-components is executed in parallel, this is broken down to pairwise execution records as well. For example, if the sub-components 1, 2, and 3 are executed in parallel, the parallel executions of $(1, 2)$, $(2, 3)$, and $(1, 3)$ are recorded. These parallelism matrices suffice to reconstruct basic information about inherent parallelism between different parts of a PDEVS model. When load balancing starts, each coordinator passes its parallelism matrix to the load balancer object at the root coordinator and re-sets its parallelism matrix entries to zero.

The load balancer has now to determine which parts of the PDEVS model should be placed on different processors, i.e. which have been executed in parallel. To do so, each parallelism matrix is interpreted as the adjacency matrix of a simple, undirected *parallelism graph*. If there is an edge between two nodes in a parallelism graph, this means that they have been executed in parallel at least once since load balancing was invoked the last time. Each sub-component should be placed on an LP where none of its parallelism graph neighbours are situated, and the number of used LPs should be minimised at the same time. This corresponds to the graph colouring problem, which is known to be NP-hard and whose exact algorithmic solutions are still too costly to work on large graphs [5]. Instead of determining an optimal solution, we implemented a depth-first search to colourise the graph. Sub-components whose nodes have been colourised with the same colour form (independent) *parallelism sets*. Each of these sets should be placed on a single LP.

Although it is now clear how the sub-components of each coordinator should be separated from each other, this does not allow a concrete assignment to the p processors that are available. To do so, the $p - 1$ parallelism sets whose separation would result in the highest performance gain need to be identified. This is done by analysing the parallelism sets of all parallelism matrices and calculating a *migration gain* for each set. The $p - 1$ sets with the highest gain should be moved to remote processors, given that the gain is positive.

We calculate the gain of separating a component i from a component j as follows: Considering the migration of j from the processor on which i is situated, let d be the average additional time that is needed to reach processor of j from processor of i (i.e., the network delay). Furthermore, let c_i and c_j be the average computing time that is needed to calculate the state transition functions of i and j respectively (including their sub-components). Now, it is possible to calculate the difference of a sequential execution (i.e., both components are hosted on one processor) and a parallel execution (i.e. component j is hosted on a remote processor). This leads to the following gain function g:

$$g(i,j) = \begin{cases} P_{i,j} \cdot c_j - 4d(P_{j,j} - P_{i,j}), \text{ iff } c_i \geq c_j + 4d \\ P_{i,j} \cdot c_i - 4dP_{j,j}, \text{ iff } c_i < c_j + 4d \end{cases} \quad (1)$$

Equation (1) is the result of some basic mathematical derivations that specifically account for the characteristics of the abstract PDEVS simulator, and the fact that the communication delay is hidden if $c_i \geq c_j + 4 \cdot d$. This simple gain function is now calculated twice[3] for each edge in the parallelism graphs. To approximate the benefits of migrating a parallelism set, all edges of its elements are considered and their migration gains are summed up. When the most beneficial parameter sets have been found, the model tree is split up accordingly. The affected components are moved to the remote processors and the simulation proceeds.

5.3. Improvements

A major drawback of the basic algorithm is that it considers neither migration costs nor the current position of each component at runtime. To overcome these problems, the gain estimation as described in equation (1) was extended to consider the number of sub-components of each component, which can be regarded as a rough approximation of migration cost.

Furthermore, a voting scheme to minimise the actual number of migrations was introduced. After splitting up the model tree, the current positions of all components are considered to decide which part of the tree should be hosted on which processor. This is realised by creating a migration matrix M of size $u \times (p-1)$, where $u \leq (p-1)$ is the number of the component sets in which the model tree was split up[4]. Each $m_{i,j} \in M$ denotes how much elements of set i are already situated on processor j. The actual mapping from model sets to processors is done by continuously choosing a maximal element $m_{x,y} \in M$, which makes processor y the host processor of all models in set x. After removing row and column of M which contained $m_{x,y}$, this procedure is repeated until the matrix is empty. This mechanism ought to reduce the number of migrations, and its effectiveness was evaluated using SIMSIM.

[3]once for migration of i, once for migration of j

[4]It is possible that there are less beneficial model sets than available processors, e.g. for models with a low granularity.

Finally, it would be desirable to relate the load balancing frequency to the dynamism of the simulated model. This requires an adaptive behaviour of the algorithm: When a model's computing requirements vary strongly over time, load balancing should be invoked more often, to enable instant reaction to changes. On the other hand, static models do not need load balancing as often, so that a too frequent load balancing could hamper the execution performance. To realise a variable load balancing frequency, the frequency was adjusted by a formula that took the results of the last load balancing invocation into account. However, as could be predicted by SIMSIM and was verified later, some conceptual problems led to oscillations (see [6] for details).

6. Results

This section presents some of the interesting results that were obtained by SIMSIM, and also illustrates how these results comply with the real performance of the algorithm. To compare predicted and real performance, the benchmark model and the algorithm have been implemented in JAMES II as well. The basic experimental setup makes several simplifications, which reduces the number of different scenarios to be observed and therefore leads to more general results.

In the following, tu_{amt} and tu_{swt} denote the time unit of application model time and simulated wall-clock time respectively. Only the relations among the time values of one dimension (tu_{swt} or tu_{amt}) are relevant. The basic experiment predicts the PDES system performance on a set of 8 fully connected, homogeneous processors (e.g., $cp = 1$ for each processor), with each connection having the same quality ($T_s = 1\,tu_{swt}, T_w = 0\,tu_{swt}$, i.e. message size does not matter). The randomly generated model tree consisted of 100 nodes, the average number of children per node was set to 4. Load balancing was repeated after 10 simulation steps. Only one state phase with a duration of 30 tu_{amt} was defined for all atomic DEVS models. The atomic models formed four exclusive communication groups (i.e., $pref$ was set to 1). Each model sent a message after a duration of 0.1, 0.55, or 1 tu_{amt}, which was randomly chosen. The response delay was 1 tu_{amt}. Finally, internal and external state transition were defined to last 10 tu_{swt} each.

6.1. Gain Borders

SIMSIM was used to investigate under which circumstances a parallel execution using the load balancing algorithm is beneficial at all. The results are depicted in figure 2. Each point in the result matrix represents two simulation runs, one in a sequential manner (i.e., on one host, without load balancing) and one in parallel. The difference of both execution times was calculated, so that positive values denote situations in which parallel simulation is faster than sequential simulation, and negative values denote the opposite.

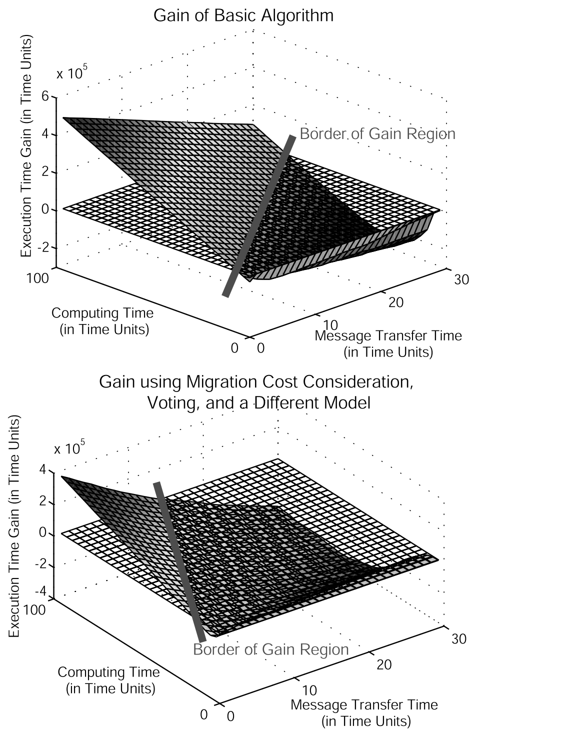

Figure 2. Gain regions for different setups

As expected, PDES is beneficial in high-granularity and low-latency scenarios. Interestingly, the *gain* border, i.e. the point from which on parallel simulation is faster than sequential simulation, is very sensitive to model characteristics. These experiments could now be used to assess the algorithm's suitability for a variety of problem subsets. Further experiments addressed scalability, performance in relation to the dynamics of model entities, and influence of model parameters on the overall number of migrations (see [6]). One could also exchange the abstract PDEVS simulator by a more optimistic version and investigate how the synchronisation (e.g., the number of anti-messages) is influenced by the load balancing scheme. To do so, one would simply have to model the optimistic simulator and count the corresponding messages.

6.2. Quality of Predictions

Figure 3 shows the difference between predicted and actual performance, measured on a two-processor system. To obtain these results, shorter simulation runs (state phase time span set to 3 tu_{swt}) were conducted using a smaller

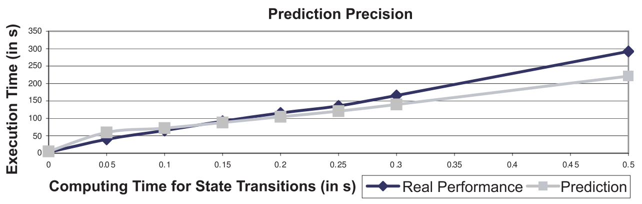

Figure 3. Performance prediction precision

model tree (40 nodes). In low-granularity scenarios, the predictions do not necessarily correlate with the experimental outcome. This comes at no surprise, since in these cases the factors that were not considered for prediction – like the performance overhead induced by the abstract PDEVS simulator itself – prevail. However, as long as the model imposes any computational effort, the performance prediction is rather accurate. It can also be seen that the precision slowly decreases in high-granularity scenarios. This can be explained by the threading overhead caused by the operating system (each processor had to maintain ≈ 20 concurrent Java threads), which was also abstracted away (see section 3.3.2). Hence, SIMSIM's predictions using the presented generic PDES model have to be considered as under-approximations. Nevertheless, a refined generic PDES system model that accounts for this overhead, even by simply multiplying an appropriate factor, can eliminate this prediction error.

7. Conclusions

This paper investigated the applicability of an established technique – the simulation of software systems – to overcome problems in PDES system development. To show that this approach is feasible, we introduced a sequential simulator for simulation systems, SIMSIM, that was used to develop a load balancing algorithm for the abstract PDEVS simulator of JAMES II. We motivated and exemplified the use of different simulation modes in development, namely interactive simulation to find the causes of unintended behaviour (e.g., oscillations, see section 5.3) and batch simulation to find general performance characteristics (e.g., gain borders for PDES, see section 6.1). Finally, we evaluated the quality of the obtained predictions.

The investigation of the presented algorithm used many abstractions, which was regarded as a drawback of mathematical models in section 2. But our point is that these abstractions are *not necessary* to get results on the algorithm's performance, as it is the case for mathematical models, but we used them to get more *general* results. With our approach, the developer can decide freely which aspects should be abstracted away, and which should not. Further research should address a more thorough analysis of the presented algorithm and ways to automate the analysis of algorithms in general. Additionally, questions about appropriate modelling formalisms for PDES systems need some consideration, as well as the refinement of the generic PDES system model to avoid prediction errors as described in section 6.2.

All in all, it can be concluded that meta-simulation *is* a feasible approach to support PDES system development. With its ability to get qualitative and quantitative data on an algorithm's performance before or during development, it can be seamlessly integrated with software development processes (e.g., extreme programming) and seems to be particularly useful for developing PDES systems that consist of many interacting mechanisms. Future applications of this approach might be an automatic selection of suitable PDES algorithms at runtime or the training of students in university courses.

Acknowledgement

This research is supported by the German Research Foundation (DFG). Prior work has been undertaken in the PDES-MAS project[5], which is supported by EPSRC research grant No. GR/R45338/01.

References

[1] V. Balakrishnan, P. Frey, N. B. Abu-Ghazaleh, and P. A. Wilsey. A Framework for Performance Analysis of Parallel Discrete Event Simulators. In *Winter Simulation Conference*, pages 429–436, 1997.

[2] R. Buyya and M. Murshed. GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing. *Concurrency and Computation: Practice and Experience*, 14(13–15):1175–1220, 2002.

[3] C. D. Carothers and R. M. Fujimoto. Background Execution of Time Warp Programs. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, pages 12–19, Washinton, May 22–24 1996. IEEE Computer Society Press.

[4] S. R. Das, R. M. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Winter Simulation Conference*, Proc. of the 1994 Winter Simulation Conference, pages 1332 – 1339, 1994.

[5] D. Eppstein. Small Maximal Independent Sets and Faster Exact Graph Coloring. *J. Graph Algorithms Appl.*, 7(2):131–140, 2003.

[6] R. Ewald. Simulation of load balancing algorithms for discrete event simulations. Master's thesis, University of Rostock, 2006.

[7] R. Ewald, D. Chen, G. K. Theodoropoulos, M. Lees, B. Logan, T. Oguara, and A. M. Uhrmacher. Performance Analysis of Shared Data Access Algorithms for Distributed Simulation of Multi-Agent Systems. In *Proc. of the 20th Workshop on Principles of Advanced and Distributed Simulation*, 2006.

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

[9] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley, 2000.

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

[11] E. Glinsky and G. Wainer. Performance Analysis of DEVS Environments. In *Proc. of AI Simulation and Planning*, 2002.

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

[13] J. Himmelspach and A. M. Uhrmacher. A component-based simulation layer for JAMES. In *PADS '04: Proceedings of the eighteenth workshop on Parallel and distributed simulation*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[14] Z. Juhasz, S. Turner, K. Kuntner, and M. Gerzson. A Performance Analyser And Prediction Tool For Parallel Discrete Event Simulation. In *UKSIM 2001: Conference On Computer Simulation*, 2001.

[15] J. Liu, D. M. Nicol, B. J. Premore, and A. L. Poplawski. Performance Prediction of a Parallel Simulator. In *Workshop on Parallel and Distributed Simulation*, pages 156–164, 1999.

[16] B. Logan and G. K. Theodoropoulos. The Distributed Simulation of Multi-Agent Systems. In *Special Issue on Agent-Oriented Software Approaches in Distributed Modelling and Simulation*, IEEE Proceedings Journal, 2001.

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

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

[19] D. M. Nicol. Scalability, locality, partitioning and synchronization PDES. In *Proc. of the 12th workshop on Parallel and distributed simulation*, pages 5–11. IEEE Computer Society, 1998.

[20] D. M. Nicol, J. Liu, M. Liljenstam, and G. Yan. simulation of large-scale networks using SSF. In *Winter Sim. Conference*, pages 650–657, 2003.

[21] K. S. Perumalla, R. M. Fujimoto, P. J. Thakare, S. Pande, H. Karimabadi, Y. Omelchenko, and J. Driscoll. Performance Prediction of Large-Scale Parallel Discrete Event Models of Physical Systems. In *Winter Sim. Conference 2005*, 2005.

[22] P. Teo, S. J. Turner, and Z. Juhasz. Optimistic Protocol Analysis in a Performance Analyzer and Prediction Tool. In *PADS*, pages 49–58, 2005.

[23] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation*. Academic Press, London, 2000.

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

[5]http://www.cs.bham.ac.uk/research/pdesmas