Simulation of Load Balancing Algorithms for Discrete Event Simulations

Diplomarbeit

Universität Rostock, Institut für Informatik

Submitted by: Ewald, Roland
Birth date: August 28, 1981 (in Güstrow)
Matr.-Nr.: 1200089
Tutor: Dipl.–Inf. Jan Himmelspach
Advisors:

- Prof. Dr. rer. nat. habil. Adelinde M. Uhrmacher (Universität Rostock)
- Dr. Georgios K. Theodoropoulos (University of Birmingham)
- Prof. Dr. rer. nat. habil. Peter Luksch (Universität Rostock)

Date of submission: April 10, 2006

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. This work illustrates the resulting difficulties with respect to the development of parallel and distributed simulation systems for general purposes. It is reasoned to what extent a prior analysis could facilitate this task. Approaches to predict the performance of parallel and distributed simulation systems are discussed. It is argued that the *simulation* of such a simulation system is a feasible approach. This claim is underpinned by implementing SimSim, a sequential simulator to simulate parallel and distributed simulation systems. Then, SimSim is used to develop a load balancing algorithm for the simulation of PDEVS models. The algorithm's performance is analysed using SimSim. Finally, the predicted performance is compared to the real performance of the algorithm when running in the simulation system James II.

Zusammenfassung

Die effiziente parallel–verteilte Simulation von diskret–ereignisorientierten Modellen ist ein herausforderndes Unterfangen. Dies liegt einerseits an der Vielzahl der zu berücksichtigenden Faktoren, wie z. B. der vorhandenen Infrastruktur und den Charakteristika des zu simulierenden Modells. Zum Anderen existieren bereits verschiedenste Algorithmen zur Lösung von Teilproblemen der parallel–verteilten Simulation, die je nach Anwendung Vor- und Nachteile aufweisen. Diese Arbeit illustriert die daraus resultierenden Schwierigkeiten bei der Entwicklung von entsprechenden Simulationssystemen. Es wird erörtert, inwiefern eine vorherige Analyse die Entwicklung von neuen Algorithmen vereinfachen kann. Verschiedene Techniken, die eine solche Analyse erlauben, werden diskutiert. Des Weiteren wird argumentiert, dass sich die *Simulation* von Simulationssystemen gut als Ansatz zur Analyse eignet. Dies wird durch die Implementierung von SimSim, eines sequentiellen Simulators für parallel–verteilte Simulationssysteme, untermauert. SimSim wird anschließend zur Entwicklung eines Load Balancing Algorithmus für die parallel–verteilte Simulation von PDEVS Modellen verwendet. Die Eigenschaften des Algorithmus werden mit Hilfe von SimSim untersucht. Schließlich wird der Algorithmus in das Simulationssystem James II eingebunden, und sein Verhalten wird mit den Vorhersageergebnissen verglichen.

CR–Classification (1998):

I.6.3[Simulation and Modelling]:Applications—Load balancing, PDEVS;

C.2.4[Computer-Communication Networks]:Distributed Systems—Distributed applications;

I.6.8[Simulation and Modelling]:Types of Simulation—Discrete event, Distributed, Parallel;

D.2.5[Software Engineering]:Testing and Debugging—Diagnostics, Debugging aids, Testing tools;

G.2.2[Discrete Mathematics]:Graph Theory—Trees, Graph algorithms;

K.6.3[Management of Computing and Information Systems]:Software Management—Software development, Software selection;

Key words: Performance Analysis, Performance Prediction, James II

Contents

Index

1 Introduction

1.1 Preface

Prediction is very difficult, especially if it's about the future.
Nils Bohr

As it has already been extensively used for the development of nuclear weaponry in the Manhattan Project, Modelling and Simulation is one of the oldest application areas of computers. Since those days, it has been applied to a broad variety of problems, from consumer behaviour in a supermarket to astronomical phenomena.

Consequently, different modelling formalisms emerged to facilitate the model specification for certain domains. The major distinction between today's models is whether they model an aspect of reality in a continuous way (e.g. by using differential equations) or by employing a discrete view. There are also hybrid approaches that combine both paradigms [65].

Discrete models are to some extent an abstraction of continuous ones, because they are restricted to a finite number of state changes within a time interval. Although there are differing views on discrete models [19], they can basically be regarded as a number of entities that notify each other of events by exchanging messages. As these so-called discrete-event models are the underlying principle of all discrete world views, this thesis is focused on discrete-event models.

With the diversification of model specifications and world views, different simulation algorithms have been proposed. Today, the list of requirements for state-of-the-art simulation systems is quite long: At first, a distributed and parallel execution is often useful, because the models become more and more complex. Besides that, interoperability or the integration of external processes are, among other issues, of importance. Especially the efficient distributed simulation leads to a number of new problems. These problems (synchronisation, load imbalance, etc.) are common for all distributed software systems, but their solution has to be suitable for the context of a simulation system. To address these problems, various algorithms have been developed by the modelling and simulation community over the past years.

This trend leads to an increasing complexity, because a simulation system can be designed as any reasonable combination of these algorithms. Moreover, different applications, execution platforms, programming languages and modelling formalisms make simulation systems even harder to compare and evaluate. Problems also arise when parts of an existing simulation system have to be enhanced or revised: A new algorithm may possibly evoke side-effects that strongly influence other parts of the simulation system.

To overcome these issues, one could test the simulation system as much as possible. Unfortunately, this approach cannot resolve all problems (e.g. the platform and programming language dependency) and is rather laborious, especially when developing parallel and distributed simulation systems, which need a set of computers for each run.

In this thesis, I will argue that a feasible approach to solve the above-mentioned problems is to apply modelling and simulation techniques on themselves: Since one of the main application areas of modelling and simulation is to gain deeper knowledge of modelled systems, modelling and simulation of simulation systems could possibly facilitate their development, testing and comparison.

To illustrate this claim, I describe the simulation-based development of a load balancing algorithm for JAMES II, a fairly complex simulation system which is developed at the University of Rostock: Firstly, I generalize an existing software for investigating certain aspects of the PDES-MAS simulation system, developed at the University of Birmingham and the University of Nottingham. This is described in chapter 4. Then, I model the relevant parts of JAMES II and use the new tool, SIMSIM, to develop a load balancing algorithm for JAMES II (chapter 5). Chapter 6 contains the analysis of SIMSIM's simulation runs, the incorporation of the gained knowledge into a real load balancing algorithm for JAMES II, and a comparison between its predicted and actual performance. Finally, I draw some general conclusions regarding the simulation approach, and give a short outlook to future research issues (chapter 7).

Before this proof-of-concept is described, chapter 2 motivates the simulation of simulation systems in a more detailed manner. Several parallel and distributed discrete-event simulation systems are briefly described and compared, so that the complexity of the discussed problem becomes apparent. Additionally, other (and similar) approaches to predict simulation performance are presented in chapter 3.

1.2 Terminology

This section clarifies the use of the most important technical terms in this thesis. For additional information, please refer to the index (page 5) or the glossary (page 108).

1.2.1 Basic Terminology

System, Model, Simulation

Obviously, these terms play a central role for modelling and simulation. Still, it is hard to choose from the dozens of proposed definitions, so the following ones have been picked regarding their usability in the context of this thesis. The American Heritage Dictionary defines a system as *"A group of interacting, interrelated, or interdependent elements forming a complex whole."* [1, p. 832]. A model is described by Zeigler et al. as follows: *"In its most general guise, a model is a system specification at any of the [system specification] levels [...]."*[107, p.29].

Thus, models can be regarded as system specifications, and their behaviour can be simulated (i.e. computed) by a simulation software (see section 1.2.4). Often, modelling is used on systems that need to be investigated, particularly if interaction with the system itself is dangerous, expensive, or impossible. One way of investigation is to *simulate* a model of the system, which means that the model's behaviour is calculated. This is usually done by a *simulation system*, on a computer. Afterwards, the simulation results are analysed, so that knowledge of the original system may be obtained.

Since systems are composed of elements, it is natural that these elements are somehow reflected in models of them as well. Those element representations will be called *entities*. If a model contains representations of a system's elements, it is possible to divide the model into several parts. This is evidently a requirement for its distributed simulation (see section 1.2.2).

Similar to the elements of a system, the entities of its model are related to each other by data dependencies. An indecomposable entity is called *atomic*. Consider the distributed simulation of a model for a vector: Each number could be seen as an atomic entity. Another entity of the model, e.g. representing the sum of the vector elements, would have a data dependency to each of the vector's number entities. Generally, the limitation of models as being composed of entities and connections between them is still very generic and subsumes a huge variety of *modelling formalisms*.

Notions of Time

The time it takes to simulate a model can be measured and used to evaluate performance issues. On the other hand, models usually incorporate a notion of time, too. To avoid confusion when using the same word in ambiguous situations, different notions of time have been identified and named:

- Physical time: The time of the system that is modelled. For instance, a model of a medieval town could cover a physical time from the 5th to the 15th century.
- Simulation time: The simulation time denotes the actual representation of time during the simulation. The start year of the medieval town simulation, e.g. 476, could be denoted as '0'. Thus, simulation time is an abstraction of physical time.

- Wallclock time: The real-world time in which the simulation is executed. Reconsidering the medieval town example, the year 1492 (physical time) could be represented as year 1016 (simulation time), and its simulation may finish after 2 hours and 20 minutes (wallclock time).

Later on, additional notions of time will be introduced (see chapter 4). A more detailed description of these terms is presented in [36, p. 27 - 28].

Speedup

The *speedup* of a simulation run is measured in relation to a preliminary run, so that the performance benefits of a certain algorithm or adjustment can be quantified. Speedup is the ratio of the preliminary run's runtime to the runtime of the modified run. It is either given as a factor (a speedup of 1.9 means that the modified run was almost two times faster) or as a percentage (a speedup of 50% means that the modified run was 2 times faster). When the speedup is less than 1.0, it is also called a *slowdown*.

Although speedup is the most common measure for the performance of a simulation system, other, more general aspects like the utility for its users could be taken into account as well (e.g. see [69]).

1.2.2 Parallel and Distributed Simulation

The adjectives "parallel" and "distributed" are often used in conjunction, but they characterise different aspects of a simulation. That is, a distributed simulation does not necessarily need to be parallel, and vice versa.

A parallel simulation processes a simulation experiment *in parallel*, i.e. it increases the performance by exploiting the "inherent parallelism" of a simulation experiment. This is useful if the execution of a *single* run takes too much time. Otherwise, if a single run can be executed fast enough but a large number of experiments has to be conducted, it would be easier to let each available processor independently execute a sequential simulation run. This way of parallelising a simulation is very powerful and easy to implement, but its use is restricted to the parallel execution of small-scale experiments. To speed up a *single* simulation run, a parallel simulation could concurrently calculate the parts of a model that do not depend on each other during a certain time interval.

On the other hand, a distributed simulation system is executed on a set of physically distributed computers. Finally, if a simulation system fulfils both requirements, i.e. it allows the parallel execution on a distributed set of computers, we speak of a "parallel and distributed simulation system". This thesis focuses on the efficient simulation of parallel *and* distributed discrete-event simulations (see figure 1.1).

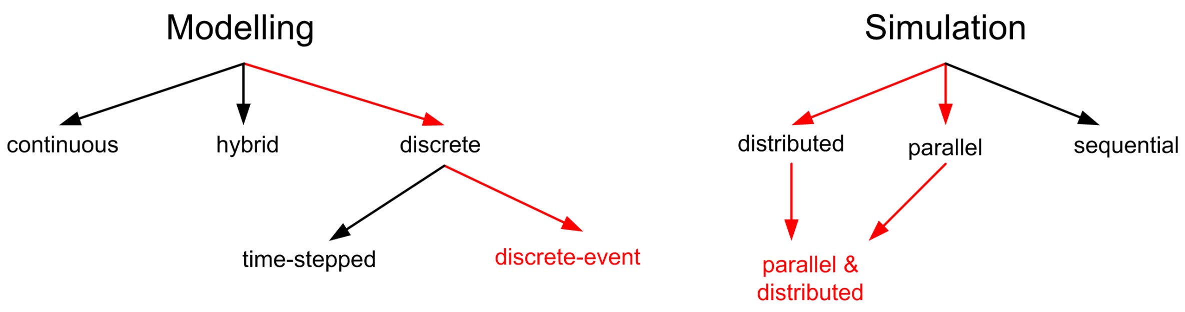


Figure 1.1: Overview: Types of Modelling and Simulation. The red arrows and labels mark the topics that are covered in this thesis.

Additional information about the difference between distributed and parallel simulations, especially regarding the computer architectures they are executed on, can be found in [36, p. 17 et seqq.].

1.2.3 Distributed Discrete-Event Simulation

As already mentioned in the preface, simulations are firstly distinguished by their modelling paradigm, whether they simulate continuous, hybrid or discrete models. Since this thesis focuses on discrete-event simulation, some related terms will be described here. Other types of discrete simulation, like time-stepped simulation (e.g. used to simulate cellular automata), can be seen as special cases of discrete-event simulation.

As Fujimoto states in [36, p. 34 et seqq.], an undistributed discrete-event simulation typically consists of state variables, a list of events, and a global clock value. The state variables represent the state the model is in.

For each simulation loop, the event with the smallest time stamp is processed and removed from the list, which may result in a number of new events (with a time stamp greater or equal) and changes to the state variables. Subsequently, the global clock value increases to the minimum time stamp of the remaining events. This procedure continues until the event list is empty.

Obviously, all events are processed in an increasing time stamp order. To ensure that the results of a distributed simulation algorithm are identical, this order has to be preserved. Otherwise, it would be possible that future events influence events in the past. This requirement is called the *local causality constraint* [36, p. 52].

To describe a distributed discrete-event simulation, Fujimoto [36, p. 40] uses Misra's concept of *logical processes* (LPs) [64]: A logical process is an abstraction of a physical process that belongs to a distributed execution. Basically, each logical process is working like the undistributed discrete-event simulation outlined before: It manages a certain set of state variables and processes a list of events. In contrast to the undistributed simulation, the processing of an event may now result in the creation of events that cannot be processed locally, because they affect state variables that are managed by other logical processes. Therefore, all logical processes of a distributed simulation form a kind of logical network to exchange events among each other. In this network, only LPs that could exchange events are connected. Like in graph theory, all LPs that have a connection to a certain LP are called its neighbour LPs, or just neighbours. Since events are exchanged via messages, the terms event and message are often used synonymously in this context. The CPU that processes an LP is named the LP's *physical processor* (PP). While an LP is executed on exactly one PP, each PP may execute multiple LPs.

Another difference between distributed and undistributed simulation is the calculation of the global clock value. Contrary to undistributed simulations, the calculation of the global clock value is not straightforward for distributed simulations. Although each logical process stores a *local* clock value, which is the minimal event time stamp in its event list, the clock values of the other logical processes are unknown.

Since the LPs ought to run in parallel as much as possible (to speed up the simulation), this leads to complications when deciding whether it is *safe* to process an event without violating the local causality constraint. To solve this problem, several approaches have been developed, which are briefly described in 2.2. Figure 1.2 sketches the basic structure of a distributed discrete-event simulation. Each LP maintains a local clock and a local event queue. Events to be processed by entities on a remote LP are sent over a network. To ensure the right execution order, each event contains a time-stamp and additional data.

In this thesis, only parallel and distributed discrete-event simulations are considered. For the sake of readability, I use the common abbreviation *PDES*, or just the term simulation, to refer to them.

Additionally, other discrete-event world views, like process orientation, can be built upon the event-oriented world view as described above. Nicol et al. characterise the event-oriented world view as follows: *"While sparse, this paradigm is general enough to support construction of richer computational models, such as process orientation."*[71, p. 233]. Therefore, it is sufficient to concentrate on the event-oriented world view in the context of this thesis, without excluding the others.

1.2.4 Simulation Languages and Systems

Finally, the term simulation system needs to be clarified. In general, as Low et al. state in [58, p. 171], one has to differentiate between simulation languages (e.g. APOSTLE [105]) and simulation executives (e.g. GTW [37]). While the latter are stand-alone systems that provide an interface for (model) applications and

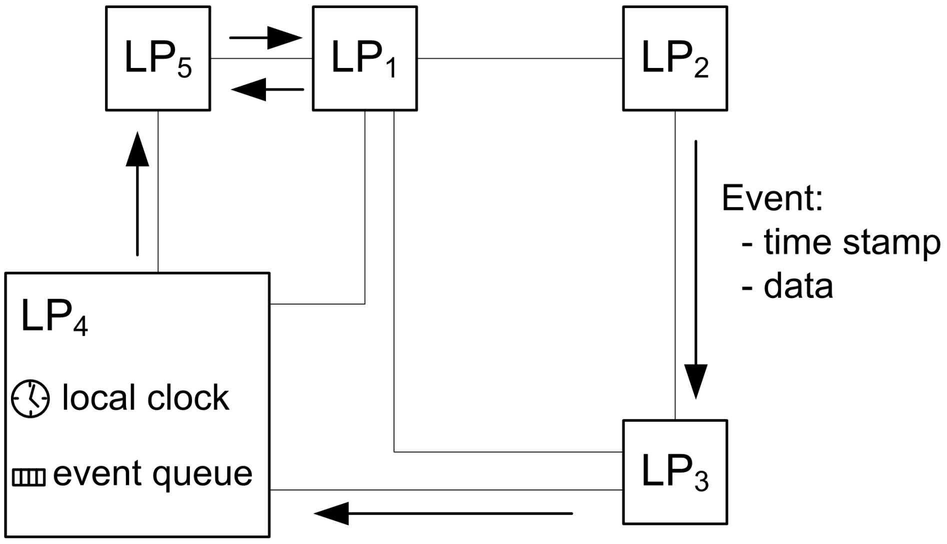


Figure 1.2: Scheme of a Distributed Discrete-Event Simulation

are comparable to libraries, simulation languages let the modeller define the model in a certain language and compile it - together with the corresponding simulation subroutines - to a single simulation program. Since this distinction is irrelevant for the actual execution, the term simulation system is used as an overall term for software systems that execute PDES.

1.3 Typesets

Unless a term has just been *emphasised* by using an italic typeset, this indicates that it is mentioned for the first time, or that there is a glossary entry providing additional information. Besides that, *"direct quotations"* from other works are set in italic letters. As usual, `monotype` text refers to command line statements or fragments of source code.

2 Background

This chapter aims at providing background information about parallel and distributed discrete-event simulation systems (PDES) in general. Before, definitions for different types of so-called DEVS models are given and explained. The DEVS formalism plays a central role for the next chapters. Then, major challenges for the efficient execution of PDES are described shortly. Finally, a brief survey of existing PDES systems is given. Owing to their relevance for the following chapters, the simulation systems JAMES II and PDES-MAS are covered in a more detailed way.

2.1 The DEVS Modelling Formalism

In this section, the Discrete-Event System Specification (DEVS) formalism will be described in detail. At first, I will describe the basic DEVS definitions (section 2.1.1). This will provide a good introduction to the modelling principles employed in DEVS. Afterwards, PDEVS, an extension to model parallelism of DEVS models, will be defined (section 2.1.2). This is a central topic, since the simulated load balancing algorithm will be tailored for PDEVS. An algorithm that can be used to simulate PDEVS models is presented in section 2.3.1. However, there are several other extensions of the DEVS formalism that will not be covered here (like DynDEVS [25], or Cell-DEVS [99]).

2.1.1 The Discrete-Event System Specification Formalism

Definition of an Atomic DEVS Model

The DEVS modelling formalism was developed by Zeigler [106] in order to provide a system modelling formalism that follows the principles of system theory. Consequently, the definition of DEVS models helps to ensure their system-like behaviour (e.g. explicit system borders, conditions for well-defined DEVS models, as described later). In principle, DEVS models can be seen as black boxes that only interact via a well-defined interface with their surrounding. Their internal behaviour is defined by a set of functions. The most basic DEVS model, the atomic DEVS model, is defined in [107, p. 138 - 139] as a tuple

$$(X, Y, S, \delta_{ext}, \delta_{int}, \lambda, ta)$$

with

X − set of possible inputs
Y − set of possible outputs
S − set of sequential states
$ta : S \rightarrow \Re_0^+ \cup \{\infty\}$ − time advance function
$Q = \{(S, e) | s \in S, e \in [0, ta(s)]\}$ − set of total states, e being the elapsed time
$\delta_{ext} : Q \times X \rightarrow S$ − external state transition function
$\delta_{int} : S \rightarrow S$ − internal state transition function
$\lambda : S \rightarrow Y$ − output function

Each atomic DEVS model may receive elements of X as input, and eventually generates outputs that are elements of Y. Most remarkably, the DEVS formalism differentiates between internal and external events. Internal events are triggered by the time advance function ta. The time advance function associates each state of the system ($s \in S$) with a time at which the system will change its state autonomously, by selecting the new state via the internal transition function δ_{int}. This internal state transition can be seen as an internal event. Before changing to the next state, λ is used to generate output.

In contrast, the *external* transition function is used to select a new state when the DEVS model receives input. The reaction of the system to this external event depends on its current state s and the time that has elapsed since the system changed to its current state. Clearly, this can only happen when the elapsed time e is in $[0, ta(s)]$. Otherwise, if $e > ta(s)$, the model would have triggered an internal event before, which explains the definition of Q. No output is generated when an external event occurs.

Zeigler et al. also formulate conditions under which a DEVS model is well-defined as a system, which means that δ_{int} does not cause infinite recursions [107, p. 141 - 142]. This so-called legitimacy of a DEVS model ensures that the model cannot change infinitely between states for which ta is 0, since this would violate the discrete-event principle of a *finite* number of events during a time interval.

Now, atomic models can be composed to more complex systems by using coupled models. A coupled model is basically a container for atomic or coupled models. It not just contains some models, but also defines their input and output relationships. From an external point of view, a coupled model has the same interface like an atomic model, so that coupled models can be nested arbitrarily. In fact, it is possible to specify an equivalent atomic model for any coupled model (i.e. its output is the same as the coupled model's output, for any input). This property is called *closure under coupling* (see [107, p. 127 - 131, 151 - 152]). Zeigler et al. [107, p. 150] define a coupled model as a structure

$$N = < X, Y, D, \{M_d\}, \{I_d\}, \{Z_{i,d}\}, Select >$$

with

X – set of possible inputs
Y – set of possible outputs
D – set of references to the DEVS submodels in $\{M_d\}$

$\forall d \in D \cup \{N\} : I_d \subseteq D \cup \{N\}, d \notin I_d$
– set of influencers for each model and the coupled model

$\forall d \in D, i \in I_d : Z_{i,d}$ is an i–to–d output translation with:

$Z_{i,d} : X \rightarrow X_d$, if $i = N$
$Z_{i,d} : Y_i \rightarrow Y$, if $d = N$
$Z_{i,d} : Y_i \rightarrow X_d$ otherwise

$Select : 2^D \rightarrow D$ – tie–breaking function

The central part of a coupled model structure is the set of its sub-models, $\{M_d\}$, and the index set D to reference it. Furthermore, data interchange between the models (including the coupled model itself, denoted by N) is modelled by $\{I_d\}$ and $\{Z_{i,d}\}$. An element $I_x \in \{I_d\}$ is a set of model references that defines all models whose outputs have to be propagated to the model M_x. Since M_x receives information from all models referred to in I_x, these models may influence the behaviour of M_x and are thus called *influencers*. This must not be confused with the term *influencees*, which stands for all models that M_x does influence by itself: $influencees(M_x) = \{M_k | x \in I_k\}$. Hence, influencees have the opposite meaning of influencers, and are implicitly given in the definition.

$\{Z_{i,d}\}$ is the output mapping of the coupled model. In general, it translates elements of the influencing sub-model's output set to elements of the influenced sub-model's input set. Without this translation, only models whose output and input sets are equivalent could be connected. However, the definition also contains two special cases: All inputs that arrive at the coupled model (N) are mapped from X to the input set of the sub-models N influences. Correspondingly, the output of all sub-models that influence N is mapped to elements of N's output set Y. All in all, these mappings define the way the information is propagated within the coupled model.

Finally, the internal state transitions of the sub-models have to be considered. For this purpose, the time of the next (internal) event (TONE) of each sub-model needs to be taken into account. The TONE of atomic models is defined by their ta function, whereas the TONE of coupled models is the minimal TONE of their sub-models. A model with a minimal TONE is called an *imminent* model. If there are more than one imminent sub-models, the $Select$ function is used for tie-breaking; it selects one of the eligible sub-models to execute its internal state transition. Since an internal transition could result in output, which then would be passed to the influencees of the model and cause them to process an external event, the $Select$ function does not only specify the order in which sub-models with equal TONE should execute their state transition. Instead, a selection could also suppress the execution of internal events for eligible sub-models that were not selected (in case they are influenced by the selected sub-model).

All in all, state transitions and input/output behaviour are completely covered by this definition. Therefore, a coupled model can be treated as an atomic model, which allows further nesting. To facilitate the specification and description of coupled models, some additional concepts are useful: Without loss of generality, it can be assumed that the input for a DEVS model contains information about the source of the input, and that the model output specifies a certain destination. Although these concepts are not explicitly stated in the given definitions, ways to identify different input sources and specify different output destinations can be easily added to any DEVS model. To facilitate the specification of coupled DEVS models, each model may define so-called *ports* for input and output[1].

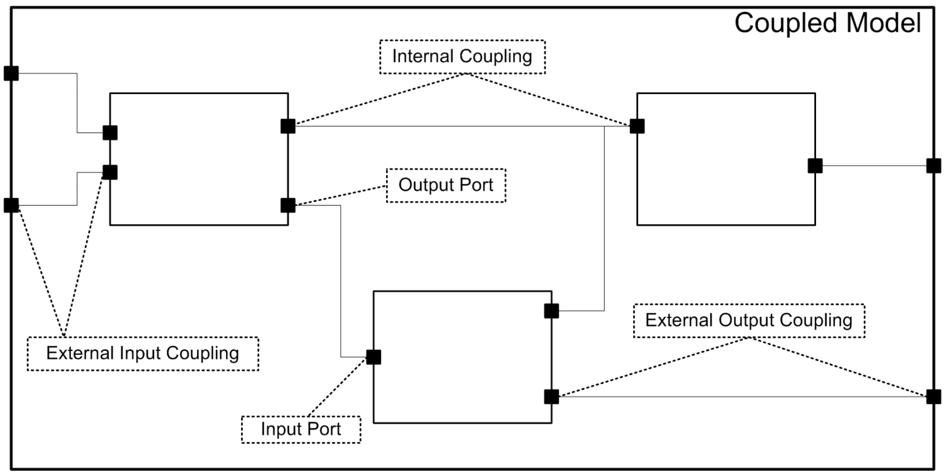


Figure 2.1: Scheme of a Coupled DEVS Model

Thus, ports can be regarded as information sockets: output ports propagate certain model outputs, whereas input ports are used to receive certain inputs. A connection between the ports of two models is called a *coupling*. Each port can be part of multiple couplings, so that broadcasts can be modelled easily. The structure of a coupled model including ports and couplings is illustrated in figure 2.1.

2.1.2 Extending the DEVS Formalism to support Parallelism

In [107, p. 261 - 284], some ways of executing the original DEVS models in parallel are discussed. However, Zeigler et al. also propose a modified definition of DEVS, PDEVS [107, p. 142 - 144], that aims at exploiting the *inherent* parallelism of a model. This means, the PDEVS formalism allows events that occur at exactly the same simulation time to be executed simultaneously. In other words, *all* imminent sub-models of a model are triggered to execute a state transition at the same time.

A major advantage of this approach is that the synchronisation problem usually attached to parallel simulation can be avoided (see section 2.2.1 for details). The definition of PDEVS models differs only slightly from the original DEVS definition in section 2.1.1.

Since all imminent models may realize a state transition at the same time, a coupled PDEVS model does not need a tie-breaking $Select$ function. This is the most important difference between a coupled PDEVS

[1]Actually, DEVS models can also be defined by explicitly using ports (see [107, p. 84 - 86]).

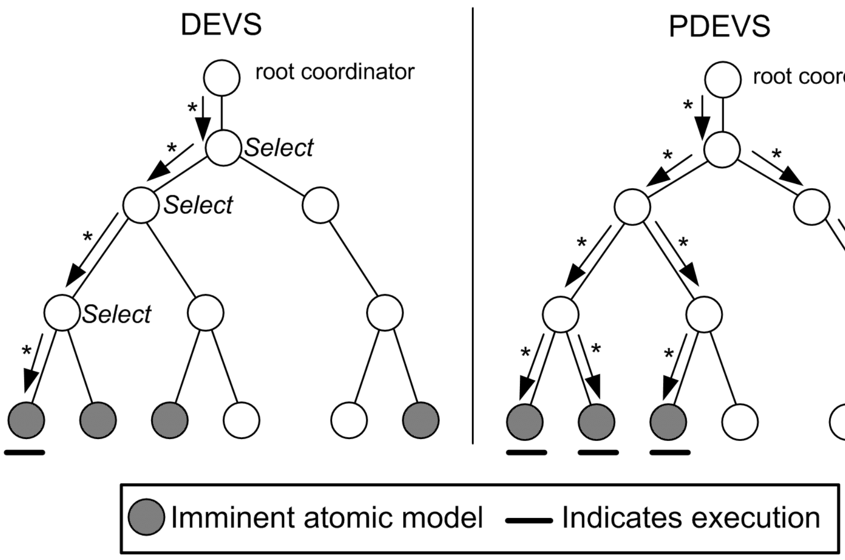


Figure 2.2: Comparison of DEVS and PDEVS. The leaves in the tree represent atomic models, which are incorporated into coupled models (i.e. their parent nodes). While the $Select$ functions of the coupled DEVS models are used to determine the imminent model to be executed, all imminents are potentially executed in parallel when using the PDEVS formalism.

and a coupled DEVS model.

In the original DEVS approach, the $Select$ function selected *one* imminent model, which could execute its internal state transition safely. All models it influenced would perform an external state transition instead. Now, this distinction is not feasible anymore. The imminent PDEVS models *cannot* just perform an internal state transition in parallel, because it is possible that an imminent model is an influencer or influencee of another imminent model. This leads to two modifications of the original DEVS model definition:

- Due to parallel execution, it might happen that a model receives *multiple* inputs (from all its imminent influencers). Therefore, input and output sets of atomic and coupled models have to be changed to *multisets*. A multiset (or bag) is a set that may contain an element multiple times (e.g. $\{a, b, b, b\}$ is a multiset, with element 'b' having a multiplicity of 3).
- It is now possible that an imminent model has to execute an internal state transition (because it is imminent) *and* an external state transition (because there is some input from other imminents). To resolve this conflict, each atomic PDEVS model defines a *confluent state transition function* δ_{con} to resolve this conflict. As for δ_{int}, a model is allowed to produce output in case of a confluent state transition[2].

A major advantage of the DEVS formalism is that it strictly separates the models from the logic to execute them. This allows to plug in different kinds of simulation engines easily. One way to execute PDEVS models is described in section 2.3.1.

Zeigler et al. also state in [107, p. 287], that the performance of a PDEVS simulation may be increased by not only executing imminent models, but models with a TONE that is *almost* the same as the minimal one. This extends the use of parallel PDEVS execution to situations where no inherent parallelism of the model can be exploited. However, this approach does heavily depend on the application and may lead to invalid simulation results if used wrongly.

[2]Often, δ_{con} is defined as $\delta_{con}(x) = \delta_{ext}(\delta_{int}(s), 0, x)$, s being the current state of the model.

It is also possible to execute ordinary DEVS models in a parallel manner, as discussed in [107, p. 261 - 284], but this thesis will focus on the execution of PDEVS models.

2.2 Challenges for Distributed Discrete–Event Simulation

This section presents some important fields of research regarding efficient parallel and distributed simulation. Actually, all of these problems can be traced back to similar problems that occur in all distributed software systems: many processes need some sort of synchronisation to interact in a meaningful way, load balancing and partitioning are considered as hard problems in the grid computing community (e.g. see [16]), and *routing algorithms* are a central topic for the networking community (e.g. see [74]). Nevertheless, the specific requirements of PDES often complicate the situations and call for suitable solutions.

2.2.1 Synchronisation

As already described in section 1.2.3, the local causality constraint leads to the difficulty that each LP has to decide whether it can process its event with the minimal time stamp.

This problem is addressed by *synchronisation algorithms*. In principle, there are two ways of ensuring event processing in an increasing time stamp order, which is sufficient to fulfil the local causality constraint: One possibility is to let each LP wait until it can be sure that processing the event with the currently smallest time stamp does not violate the increasing time stamp order, i.e. it is sure that it will not receive another event with a smaller time stamp. This approach is called *conservative* synchronisation.

Another way of synchronisation is to let each processor process the events in its event list as fast as possible. Obviously, this can lead to violations of the increasing time stamp order, and thus of the local causality constraint. The key idea behind this so-called *optimistic* synchronisation is to detect these violations and correct them. To do so, the processors that processed events in the wrong order need to *roll back* to the state before the violation occurred. The violation of the time stamp order is detected when an LP receives a so-called *straggler event*. This is an event with a time stamp less than the local clock value. When this happens, the LP needs to be rolled back to the time stamp of the straggler event.

Both synchronisation approaches bring some new difficulties: For conservative simulations (i.e. simulations that employ a conservative synchronisation scheme), the synchronisation algorithm has to define the situation in which an LP can be sure that no event with a smaller time stamp may reach it. Therefore, LPs have to somehow notify each other about the minimal event time stamps they could generate in the future. Usually, they do so by sending messages to their neighbours. A main goal of conservative algorithms is to minimise the amount of this notification messages. This is very important, because each LP has to be sure that *none* of its neighbours will generate an event that violates its time stamp order. This could lead to a large communication overhead.

Furthermore, each notification should ensure safe event processing for as much time as possible. Model-specific information can often be used to calculate an LP's *lookahead* for a neighbour, which is the smallest time stamp of a new event the LP could possibly send to it. In fact, a good lookahead calculation is very important for conservative execution, because a large lookahead increases the chance that more than one LP can process its event list.

A small lookahead also leads to another problem: dead-locks. A dead-lock is a situation, in which every LP needs a notification by at least one other LP to proceed. All LPs are blocking each other, and the simulation does never complete. Of course, this situation has to be avoided. There are several algorithms for dead-lock detection, and also techniques to entirely prevent them.

Like the conservative simulation approach, an optimistic simulation introduces several new challenges. The basic idea of optimistically executing all LPs and letting roll-backs solve the rest is called the *Time Warp* algorithm [49, 34], which brings some serious performance problems.

Firstly, the execution of roll-backs requires the storage of additional information, so that a former state can be restored. Consequently, a simulation with a large state may need huge amounts of storage. This problem can be solved by computing the *global virtual time* (GVT), which is the minimum time stamp of any event in the simulation. The idea is that no LP can be rolled-back to a state before the GVT, because there is no potential straggler event with a small enough time stamp. GVT calculation by itself is not trivial, but there are several algorithms to compute this value (e.g. by Mattern [62]). Afterwards, the *garbage (or fossil) collection* of each LP can remove all information concerning states before GVT.

Another major problem is that roll-backs often cascade. An LP that received a straggler event and was rolled back also has to invalidate all events it sent wrongly to other LPs. Often, this causes other LPs to roll back as well, and so on. This can greatly hamper the performance of the simulation. Although this phenomenon cannot be avoided in principle, it is possible to regulate the degree of optimism for each LP, so that faster LPs stop at some point and thus reduce the danger of being rolled back (see descriptions of the Moving Time Window approach [91], or its extension to the Breathing Time Warp approach [95], in section 2.3.3).

Nevertheless, optimistic synchronisation has the big advantage of actually exploiting as much “inherent parallelism” as possible, whereas conservative simulation tends to be easier to implement and test. Interesting results are also achieved by combining conservative and optimistic approaches [23, 6]. A good overview of the most important synchronisation algorithms can be found in [36].

2.2.2 Granularity

The granularity of a simulation can be understood as the ratio of the actual event processing effort to the overhead of sending it from one LP to another. This measurement is quite important, because it limits the possible speedup of a distributed execution. If events can be processed quite fast, but need a long time to be sent to another LP (i.e. the granularity is low), a distributed simulation could even execute much slower than a sequential one.

Obviously, the granularity not only depends on the simulation experiment, but also on the hardware the PDES system is executed on: execution on a multiprocessor computer or on computers connected by a very fast network could drastically decrease the overhead of sending an event.

Another way of increasing the granularity is to coalesce all events an LP sends to a certain neighbour. By this bundling, the overhead of sending a single event is reduced. For appropriate models, this approach may greatly increase the granularity, and thus the possible speedup. For example, Wonnacott et al. report a speedup of as much as 80% ([105], see section 2.3.3). On the other hand, this approach is not feasible for experiments where few events are sent to a large number of neighbour LPs.

2.2.3 Partitioning and Load Balancing

Distributed simulation requires some additional mechanisms to ensure efficiency throughout the execution. At first, the model (i.e. its state variables) has to be distributed over all available LPs. Consequently, it has to be partitioned into several parts first. This might sound trivial, but a *partitioning* algorithm has to optimise the partition concerning two objectives:

- Minimal Communication: The amount of data that needs to be exchanged between the LPs should be as small as possible.
- Load Balance: Each LP should manage a part of the model that corresponds to its computing power.

Both objectives are quite important, but it is very hard to optimise them in combination: For example, a partition that assigns the whole model to one LP would evoke minimal communication (nil, in fact), but this results in the largest load imbalance that is possible. On the other hand, it is possible to generate a perfectly balanced partition in which every LP gets its appropriate share of the model, but most likely with a huge communication overhead.

This consideration leads to a basic question: How much load imbalance is acceptable in order to save a certain amount of communication load (or vice versa)? The answer to this question depends very much on the granularity, because communication overhead hampers the execution of low granularity simulations quite strongly, whereas load balance may be more important for simulations with a high granularity. Furthermore, a large load imbalance could increase the number of roll-backs during an optimistic simulation: too lightly loaded LPs could receive an increasing number of straggler events from too heavily loaded LPs. The granularity itself depends on the model and the hardware used for the simulation (see section 2.2.2), so that a partitioning algorithm has to take all these aspects into account.

Generally, partitioning could be seen as an optimisation algorithm that searches a partition p for which the function

$$partition_{objective}(p) = factor \cdot imbalance(p) + (1 - factor) \cdot communication(p)$$

is minimal. By choosing $factor$ from the interval $[0, 1]$, the importance of load balance can be weighted against the importance of communication avoidance. Exemplary mathematical definitions for functions to calculate imbalance and the amount of communication can be found in [29, p. 35 - 36].

However, most partitioning algorithms search for approximate solutions of the *NP-hard* partitioning problem, as it is known in theoretical computer science (a good survey can be found in [31]). Hence, they try to find an optimally balanced partition with minimal communication between the LPs, as opposed to approaches that optimise *both* objectives.

Another research field deals with the development of *load balancing* algorithms for PDES. Since models may dynamically change their structure and computational demands, the partitioning of a model is not sufficient to ensure an efficient execution for longer simulation runs. Therefore, the simulation system has to adapt to changing communication and computation requirements, usually by moving parts of the model from overloaded LPs to underloaded ones. This is done by executing a load balancing algorithm that determines which model part has to be moved whereto.

The development of a load balancing algorithm brings several new challenges: Firstly, the data that is needed to re-partition the model has to be gathered. This is not trivial by itself, because each measuring and calculation decreases the performance of the actual simulation. Secondly, the simulation system has to support the migration of model parts during the execution, which complicates the implementation (e.g. events may have to be forwarded to the LP which received the migrated model parts). Thirdly, the load balancing algorithm has to decide at which frequency an analysis of the collected runtime information makes sense: If the load balancing frequency is too high, the simulation performance is unnecessarily hampered by the execution of the algorithm. Is the frequency too low instead, the system is not able to react on sudden changes within the model.

Due to this lot of problems and its rather tedious implementation, a load balancing algorithm is an ideal candidate to be tested via a simulation beforehand. A brief survey of load balancing techniques for PDES is presented in section 5.1.

2.2.4 Further Challenges

As already mentioned at the beginning of this section, all distributed system requirements retain their validity and importance when developing distributed *simulation* systems. The following problems do not play a very central role for execution of a PDES in general, but may need consideration when developing a simulation system anyhow. This list is not exhaustive.

Scalability

Scalability can be defined as the ability of a PDES system to cope with increasing problem size and an increasing amount of available hardware. In [68], Nicol investigated how model characteristics and partitioning affect the scalability of a PDES system. He also managed to define formal scalability requirements for a model, an architecture, and a PDES system (see section 3.1), but these requirements solely define upper and lower bounds. Moreover, they are hard to evaluate for most systems, so that scalability usually needs to be tested by hand.

Routing Algorithms

As already mentioned, routing algorithms are of particular interest for the networking research community. Nevertheless, routing algorithms may also need additional consideration when developing distributed simulation systems. For example, if a simulation system subdivides the used LPs to fulfil different tasks. This is the case for the PDES-MAS simulation system, where some specialised LPs form a logical network to store a certain set of state variables efficiently. Other LPs have to generate special queries in order to access these variables. Since queries may also access *sets* of state variables at once, they need to be routed efficiently within the logical network (see section 2.3.2 for details).

Interaction with External Processes

Sometimes, simulations are used as a testing tool for hard- or software. Since analytical simulations should usually run as fast as possible, specialised synchronization mechanisms are needed to integrate external processes with a PDES system. For example, the wallclock time, in which the behaviour of an external process can be observed, has to be converted to simulation time. This allows to generate meaningful simulation events for the representation of the external process within the model. Otherwise, interaction between an external process and its (simulated) environment would depend strongly on the simulation speed, which in turn depends on the hardware etc., and thus would not be valid.

Furthermore, simulation algorithms may have to be adapted to support waiting for external processes (e.g. see [45]).

Interoperability and Reusability

Since the development of valid computer models and simulation systems is quite expensive, efforts to increase the reusability and interoperability gain more and more importance. The most popular approach is the High Level Architecture [22], developed for the U.S. Department of Defense.

The idea behind HLA is that simulation systems implement a pre-defined interface. Then, they are able to interact over the Runtime Infrastructure (RTI), which provides services of a distributed operating system and some additional functionality to support external processes and data collection mechanisms. In this way, simulation systems can be coupled and synthesised to form new applications. It is also possible to define custom data types (using HLA Object Models) that can be exchanged between different simulations.

However, these requirements increase the complexity of simulation system development, because each stand-alone simulation system needs an additional interface that has to be tested and validated.

Execution on a Grid

The *grid* technology aims at providing computing resources as easy to use and flexible as the usage of the power grid [26]. This means, large applications are executed on a *virtual* supercomputer, which is actually a network of (normal) computers. This virtual supercomputer may be used by many users from remote locations, and with very different types of applications.

Evidently, the execution of an application will be distributed over the available network nodes. This is not a new idea, but grid software usually combines several mechanisms like resource management, service discovery, and user management. Basically, grid software works as a distributed operating system for the concurrent execution of many distributed applications.

The key advantage as opposed to *literal* supercomputers is the economical extensibility and setup of grid systems. Older computers may be integrated into the grid, as well as *clusters* or workstations with specialised hardware. All resources are managed by the grid system, which also ensures that only authorised users may start applications. Moreover, grid software has to provide mechanisms to recover from computer failures on certain nodes, so that a stable execution of an application is granted [46].

Although grid systems should facilitate the development of distributed applications, several considerations regarding the usually high latencies between grid nodes may have to be taken into account when developing algorithms for PDES systems [50].

2.3 A Brief Survey of Distributed Discrete-Event Simulation Systems

In this section, some sample simulation systems for PDES are introduced and described. This should illustrate that the problems mentioned in section 2.2 are indeed very common, and that several very different solutions have already been implemented. A more detailed survey of some PDES systems can be found in [58].

2.3.1 James II

James II, the Java Agent Modelling Environment for Simulation, was developed by Himmelspach et al. [45]. It is a simulation system that aims at being highly flexible and efficient at the same time. It supports different modelling formalisms (e.g. *Cellular Automata* or DEVS) and a variety of simulation components to execute them. To date, James II mainly supports the efficient execution of different kinds of DEVS models.

Additionally, James II allows to choose the set of simulation components that suit best to the model characteristics. For example, a certain simulation algorithm or *event queue* may have a huge influence on the performance.

Other research aspects concern the reuse of model specifications [86] and the application of James II to various application areas (e.g. agent testing [39], intelligent tutoring systems [59] and bioinformatics [25]). All in all, the system is quite complex and integrates a variety of algorithms and simulation tools.

In the following, the aspects of James II that are relevant for the next chapters will be described in detail. At first, an algorithm that simulates PDEVS models is outlined. Afterwards, I will give some relevant technical details about the way distributed parts of the simulation communicate with each other.

The Abstract PDEVS Simulator in James II

The PDEVS simulation algorithm by Chow et al. [20], which is used throughout this thesis to simulate PDEVS models, is called the abstract PDEVS simulator. A sample implementation is also given in [107, p. 284 - 287]. The James II implementation described here differs in some details from the original one, as it minimises the communication overhead. This will be explained later.

The abstract PDEVS simulator consists of a hierarchy of individual components. Each atomic PDEVS model is associated with a *simulator* component, and each coupled PDEVS model will be executed by a *coordinator* component. The simulators and coordinators reflect the structure of a PDEVS model (as can be seen in figure 2.2), and thus form a tree, which is called the *abstract simulator tree*. The abstract simulator tree is similar to the tree of PDEVS models, except that each node represents a simulation component (i.e. a simulator or coordinator) instead of a coupled or atomic model. A special component in the abstract simulator tree is certainly the *topmost coordinator*. Its parent is the *root coordinator*, which controls the overall PDEVS execution. All in all, there are three different components that form an abstract simulator for PDEVS: the PDEVS simulator, the PDEVS coordinator, and the PDEVS root coordinator.

The components of an abstract simulator tree are only communicating along its edges. This means, all communication is *strictly hierarchical*. Simulators only communicate with the coordinator of the coupled model their atomic model is part of. Coordinators only communicate with their parent coordinator and components that simulate *direct* sub-models. A model is only accessed by the component it is associated with. The abstract PDEVS simulator uses five types of messages, which are listed in table 2.1.

At first, the root coordinator creates an i-message for the topmost coordinator, which then propagates it along the edges of the tree, towards each simulator (i.e. each leaf node in the abstract simulator tree). A simulator returns an i-message with the time advance function value of its associated atomic model. Hence, a coordinator is able to compute the minimal ta value of its coupled model's sub-models. This minimal ta value can be seen as the ta value of the coupled model. It is propagated to the parent coordinator, which repeats this procedure. When the last answer to the i-messages reaches the topmost coordinator, i.e. it

Name	Use
i-Message	Initialisation of the simulation
*-Message	Activation of an imminent model
x-Message	Sending input to a model
y-Message	Sending output of a model
done-Message	Notification that a state transition is done

Table 2.1: Message Types used by the Abstract PDEVS Simulator

has knowledge about the ta values of its associated model's direct sub-models, it returns an i-message to the root coordinator and the initialisation is finished.

Then, the root coordinator triggers a simulation step by sending a *-message (*star* message) to the topmost coordinator. The topmost coordinator will forward the message to all of its child components that have returned minimal ta values. Correspondingly, the *-messages will be propagated by all coordinators towards the simulators of the imminent atomic models.

Now, the simulators of the imminent models have to let their models perform a state transition. However, at this point, a simulator is not able to decide whether it should let its model perform an internal or a confluent state transition. This is because all imminent models change their state *in parallel*, which makes it possible that an imminent's output influences another imminent model. Obviously, the latter would have to perform a confluent state transition instead of an internal one.

Therefore, the actual execution of a simulation step is divided into two parts. When receiving a *-message, each simulator will only execute the output function on the current state ($\lambda(s)$). This is safe, since the PDEVS definition allows output for δ_{int} as well as δ_{con}. Then, the model output will be sent as a y-message to the simulator's coordinator.

The main task for the coordinators is now to map the model outputs they received via the y-messages to model inputs, which then can be sent to the simulators as x-messages. This might sound trivial, but it is rather complicated. In general, a model's output may have to be propagated through the whole tree. Due to the structure of a coupled model (see figure 2.1), an atomic model might produce output that will be propagated over the external output coupling of its coordinator. Thus, each output may have to be sent 'upwards' in the abstract simulator tree. At the same time, when these outputs are eventually coupled with input ports, they can get 'downwards' in the tree again, until they end up as input for an atomic model.

All couplings in the PDEVS model define the destinations of a produced output, but each coordinator has only local knowledge of them, i.e. regarding the direct sub-models of its coupled model. To solve this problem in JAMES II, each coordinator waits until all y-messages from sub-components that were triggered by a *-message have arrived. When this is finished, the outputs can be sorted, and a y-message with all outputs for the external output coupling is sent to the parent coordinator. Then, the coordinator needs to wait for external input. In JAMES II, all inputs arrive in a single x-message. After including the external inputs to the internal inputs caused by outputs of its sub-components, the coordinator is able to allow each component to proceed with the execution. It does so by sending an x-message with all inputs to each component handling an imminent model, or a model that is influenced by one. If an imminent model is not receiving any input, an empty x-message is sent to its associated simulator component.

Finally, the simulators may execute the second part of the simulation step. When receiving an x-message and the model is not imminent, δ_{ext} is executed. If the model *is* imminent, either δ_{int} (the x-message is empty) or δ_{con} (the x-message is *not* emtpy) are executed instead.

Meanwhile, each coordinator waits for the done-messages from the components it sent an x-message to. When all components have signalised that they are done, a done-message is sent to the parent coordinator. Thus, the done-messages get propagated until they reach the root coordinator. Piggy-backed to a done message, each simulator and coordinator will send their new TONE values, so that the new imminents can be identified.

In contrast to the other coordinators, the root coordinator will not propagate the done-message. Instead, it will either quit the simulation (when all TONE values are set to infinity) or send a new *-message to the topmost coordinator.

Thus, all in all, the abstract PDEVS simulator executes each simulation step in five phases:

1. Activation of all imminent atomic models by propagating a *-message top-down through the abstract simulator tree
2. Execution of all imminent atomic model's output functions
3. Propagation of the outputs bottom-up and top-down to all influenced atomic models
4. Execution of the state transition, either δ_{int}, δ_{ext}, or δ_{con}
5. Notification of simulation step completion by sending a done-message bottom-up

The main difference between the implementation presented in [107] and the solution used in JAMES II is the way the delivering of a model's output is realised. In general, it is not necessary for a coordinator to wait for the outputs of all sub-components, because each incoming y-message could be processed independently and sent to the parent coordinator if necessary. Correspondingly, each coordinator could process the external input. However, this results in a larger number of exchanged x- and y- messages, and thus in a larger communication overhead. In JAMES II, each coordinator aggregates all messages, so that in any case only one x-message and one y-message are transferred between parent and child model.

A major drawback of using the abstract PDEVS simulator is that the communication effort for one simulation step is quite large: in the worst case, if all atomic models are imminent, four messages are transferred for each edge in the tree (a *-, x-,y- and done-message). This could still lead to a huge overhead when distributing the simulation. Consequently, several other efficient algorithms to execute a PDEVS model have been developed. One idea is to flatten the abstract simulator tree [53]. This approach lets the root coordinator communicate directly to all simulators, which avoids latencies that occur when messages need to be propagated within a hierarchy.

Communication between Simulation Parts

Since JAMES II is written in Java [48], which is commonly known as providing a good support for network applications, one could think that the communication between simulation parts is merely a technical detail. For example, Java programs may use Remote Method Invocation (RMI) to access Java objects on remote hosts. However, load balancing requires the ability to *migrate* parts of the simulation at runtime. By 'just' using RMI, this is not possible, due to the way it enables the message invocation.

An object that should be accessible by remote applications via RMI has to be the *sub-class* of a basic RMI object, like `UnicastRemoteObject`. Then, a so-called *skeleton* of the remote object is placed on the host from which it is accessed. The method calls to the remote object are handed over to the skeleton, which then forwards them over the network to the *stub* of the object. This stub serves as a proxy object (corresponding to the proxy pattern [38, p.207 – 217]) for the actual object. This means, it implements the same interface as the original object, and to call one of its methods will cause it to call the corresponding method of the object. Likewise, parameters and return values are forwarded. The stubs and skeletons of all remotely accessible objects have to be generated by the RMI compiler, `rmic`, beforehand.

To migrate an object from one host to another, it has to be *serialisable*. In the context of object oriented network programming, serialisation means the conversion of an object to an array of bytes. After sending an object's serialised representation over the network, the remote host is able to de-serialise it and to obtain the original object. Unfortunately, remote objects (i.e. objects that are remotely accessible) *cannot* be serialised like ordinary objects. Instead, the serialisation of a remote object results in its serialised stub, which may then be migrated. The *real* remote object, however, remains on the same host.

One workaround would be to store the remote object's data within a serialisable object and to send this object over the network. Afterwards, the new host of the object could create a new object and initialise it with the data object it received. The drawback of this approach is that the object re-creation might cause a certain overhead. Moreover, this method is quite complex to implement, as each object to be migrated has to provide additional methods for retrieving and setting the object's state.

In JAMES II, this problem is circumvented by an abstract communication layer, which also uses a proxy pattern [38, p.207 – 217]. Coordinators and simulators are not communicating directly with each other, but by using proxy objects. A proxy object stores the reference to a (remote or local) reference object that is able to access the object in question directly (see figure 2.3). To implement access to the reference object, one could even fall back on RMI or any other communication framework.

In case of a migration, a simulation component can now be serialised and transferred, like all of its proxies to access other components. Only the relatively small and simple reference objects that allow remote objects to access the migrated component need special treatment. When using RMI, these objects have to be re-created on the new host, and all existing references to their old instances have to be updated.

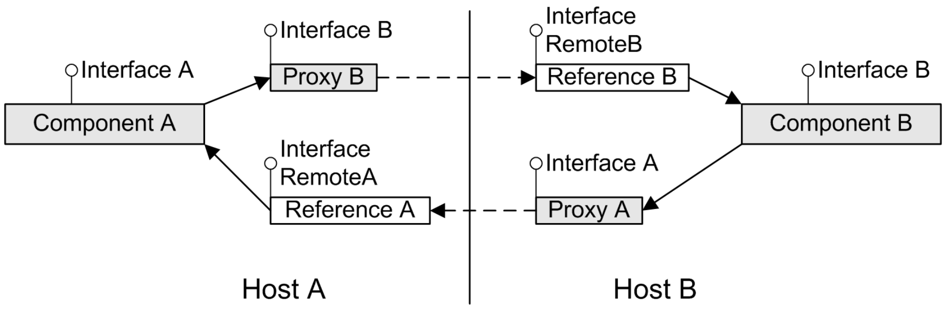


Figure 2.3: Communication using intermediate Proxy and Reference Objects. Serialisable objects have a grey background. So, a component A and all of the proxies it uses can be moved to another host. Only the proxies used by remote components to access A still point to its old reference object, and thus have to be updated.

2.3.2 PDES-MAS

The PDES-MAS simulation system (Parallel Discrete-Event Simulation of *Multi-Agent Systems*) [57] was developed by Logan, Theodoropoulos et al. to simulate systems of multiple situated agents efficiently.

In this context, it is sufficient to regard agents as complex and partly autonomous objects. A situated agent is an agent that is embedded into an environment, which is part of the simulation. Usually, situated agents are able to move within this environment and do not have global knowledge about its state. However, they are able to sense some aspects of their surrounding within a certain range. In a multi-agent system, many agents are embedded into the same environment and interact with each other. Since even one agent might be rather complex and employ artificial intelligence methods, multi-agent systems may be extremely complex to develop. The PDES-MAS simulation system aims at facilitating the development of multi-agent systems by providing an efficient platform to simulate them. This helps to gain deeper understanding of the systems, so that design decisions can be evaluated, and already implemented agents can be tested easily. Popular examples for multi-agent systems are Tileworld [80], Boids [85], or the Robocup Simulation League [84].

One problem of simulating multi-agent systems is that the state of the simulated model is usually quite large. Employing optimistic synchronisation techniques, as it is the case in PDES-MAS [55], even aggravates this issue, because some state saving mechanism is needed to support roll-backs (see section 2.2.1).

The state of the simulation model can be divided into two parts: On one hand, there are private state variables for any entity of the simulated system (e.g. an agent). These variables cannot be accessed directly by other entities. Since only one entity is able to access these variables, they should be stored on the same host that executes the entity, so that a communication overhead is avoided.

On the other hand, each entity might also incorporate public state variables that can be requested by any other entity. The set of all public state variables is called the *shared state* of the simulation, because the *shared state variables* (SSVs) it contains may be accessed by any entity of the system.

From the object-oriented programming perspective, the shared state variables are the result of a decomposition of (public) objects. To identify an SSV, each SSV gets a unique identifier, its SSV ID. Additionally, each SSV is associated with a certain *SSV type*, which represents class and field this SSV is an instance of. For example, the type of an SSV that stores the X-position of a box could be named `Box-XPosition`. As an example of a shared state, one could think of everything that is visible on a Robocup game's virtual football pitch, whereas the plans of the competing football agents are private, and are therefore not part of the shared state.

To manage the shared state efficiently, the concept of LPs is extended: PDES-MAS differentiates between agent logical processes (ALPs) and communication logical processes (CLPs). An ALP is a process that executes an agent and holds its private state. In contrast, a CLP manages a portion of the shared state. CLPs communicate with each other along the edges of a binary tree, the CLP tree. Each ALP is connected to a leaf node of the CLP tree, which is called its server CLP. The server CLP is the ALP's interface to the shared state of the simulation.

In general, the CLP tree can be seen as a distributed database system that is tailored to the needs of optimistic distributed simulation. This means, the CLPs do not only store the shared state, but they are also responsible for state-saving, as well as roll-back detection and execution. The structure of the CLP tree and the ALPs connected to it is shown in figure 2.4.

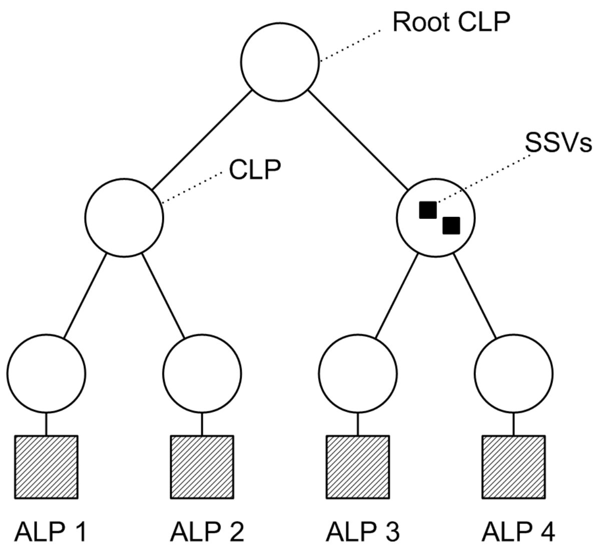


Figure 2.4: Structure of the CLP Tree

To access the shared state, an ALP creates a query and sends it to its server CLP. Clearly, different operations have to be supported. First of all, an ALP may need to read or write a single SSV. Hence, it issues a query that indicates the type of the access operation (i.e. read or write) and attaches the ID of the SSV it requests to access. The server CLP will resolve the query, potentially by propagating it through the CLP tree, and eventually return a reply message to the ALP. A query will be propagated if the current CLP is not managing the SSV to be accessed.

Now, one challenge is to identify the CLP managing the SSV to be read or written, so that the query can be routed to the CLP along the edges of the CLP tree. This problem gets complicated when so-called *range queries* are supported as well: Basically, a range query resembles a selection statement of a rational database system. It requests the IDs of all SSVs that are of a certain SSV type and that have values within a certain range. This is particularly useful for the simulation of agents, because agents, as already mentioned before, usually do not have complete knowledge of their environment. Therefore, a situated agent has to 'sense' its environment to explore it.

For example, an agent does not necessarily know all SSVs that represent box objects within its environment, nor does it know their SSV IDs. To get information about the boxes within its range (i.e. its direct surrounding), the agent issues a range query for the range it is interested in. As opposed to range queries, queries concerning only a single SSV are named ID queries.

To solve the routing problem, two different routing algorithms have been proposed and evaluated [30]. Besides the routing problem for range and ID queries, the load of the CLPs has to be managed as well. This is done by an algorithm developed by Oguara et al. [73], which migrates the SSVs with respect to their access history. Using this mechanism, agents that frequently access a certain SSV may eventually 'drag' this SSV to their server CLP.

All in all, the PDES-MAS system is a simulation system that employs a set of sophisticated and specialised mechanisms to enhance the simulation performance for a certain class of models.

2.3.3 Other PDES Systems

Now, several other parallel and distributed discrete-event simulation (PDES) systems and their main characteristics are presented briefly. The descriptions focus on each PDES system's particular solution for the main challenges of PDES (as already described in section 2.2). All PDES covered here have been detailed in several research papers - which is, in fact, one of the reasons they have been selected. Furthermore, only PDES systems that are applicable for general simulation purposes have been considered. Other systems, which are restricted to simulate only particular types of models, were left out.

For these reasons, the selection of PDES systems presented here is by no means representative. Nevertheless, the examples show the commonalities and differences between the described systems quite clearly. These will be summed up at the end of this section.

APOSTLE

The APOSTLE simulation system was developed by Wonnacott et al. [105] in 1996. It is built upon an optimistic simulator [104] and is actually a simulation language.

For synchronisation, the Breathing Time Bucket (BTB) protocol was used [95]. This approach uses the notion of an LP's *event horizon*. At first, each LP processes the events in its event queue and stores the newly created events locally. It proceeds until the minimum time stamp of the newly created events is less or equal to the next event in its event queue (events are processed in increasing time stamp order). When this happens, the *local* event horizon (LEH) of the LP is set to the minimum time stamp of the newly created events. Then, a protocol similar to those for GVT calculation can be used to calculate the *global* event horizon (GEH), which is the minimum of all local event horizons. All events that have been processed so far *and* have a time stamp not larger than the GEH are safe and can be *committed*. This means, they cannot be rolled back and all of their state information, which was saved for the case of a roll back, can be deleted. Only events with a time stamp between the global and the local event horizon have been processed in an optimistic manner and may be rolled back. Now, each LP sends out the newly created events that have been generated by *committed* events and are therefore safe to process. Some LPs might roll back towards GEH, because the events they received indicate a violation of the local causality constraint. Then, the same procedure continues.

The research on APOSTLE regarding execution performance focused on the impact of granularity. The granularity of the simulation was reduced by coalescing a number of events before sending them, in one message, to another LP. Wonnacott et al. conclude that *"APOSTLE's granularity control mechanism will only give runtime reductions when, as a result of mapping multiple objects to the same LP, there are a reasonable number of simultaneous events for the LP itself."* [105, p. 120].

Thus, granularity control, for which Wonnacott et al. show that it is able to reduce the simulation runtime by as much as 80%, depends on the model's characteristics and the ability of partitioning and load balancing mechanisms to provide a helpful object-to-LP mappping. Several simulation experiments with different characteristics have been executed, for example a simulation of the Colombian health care system (as a queueing network) and a hardware simulation.

GTW

One of the most popular simulation executives in research is the Georgia Tech Time Warp (GTW) simulation executive by Fujimoto et al. [36, 37, 24].

At first, GTW was developed for cache coherent shared-memory multiprocessors. On a shared-memory multiprocessor system, multiple processors are connected via a bus and have access to a central memory. To avoid bottlenecks, each processor may be equipped with a cache to store data that is accessed frequently. In a cache coherent multiprocessor, each processor has a cache that ensures data coherency. This means, that a processor's changes to data in its cache are immediately available for all processors accessing the same data. In other words, cache coherency makes caches transparent for the execution of concurrent processes that are accessing the same memory.

GTW employs an event-oriented world view by providing a message interchange system between LPs. One goal of the GTW development was to create a high-performance simulator for models with a low granularity (i.e. with many events that can be processed without much computational effort). In a low granularity scenario, communication overheads have a huge influence on the overall system behaviour, since the processing of events can be done quite fast.

The use of shared memory facilitates some problems of distributed simulation. For example, the GVT computation can be implemented straightforward using locks on global variables. Furthermore, the message passing can be implemented quite efficiently. Anyhow, GTW may be executed with different underlying communication layers, like *PVM* or *MPI*, as well [54].

GTW supports different optimistic synchronisation methods.This is achieved by allowing the creation of so-called blocking I/O events. Originally, blocking I/O events were used to synchronise the simulation with external processes, which usually involves input and output.

The key property of these events is that they must not be executed optimistically. Instead, they may only be processed at GVT, which makes sure that they will be committed. This allows, for example, to output valid interim results during simulation runtime.

A blocking I/O event causes the LP to stop processing any event until the GVT reaches its time stamp, whereas the non-blocking event, although not executed until GVT, allows the LP to process other events meanwhile. Different synchronisation schemes can be implemented by generating additional blocking I/O events.

In [24], Das et al. also describe a phenomena named *GVT thrashing*. It occurs when an LP has advanced far ahead from GVT, and thus has filled the available storage with its event history. When no more memory is available, the LP will initiate a GVT calculation, so that its garbage collection can free some memory thereafter. When the garbage collection is not able to free a large amount of memory, since the GVT is still too small, the same LP will initiate GVT calculation over and over again. Obviously, this hampers the execution of the other LPs and thus the advance of the GVT, which results in a bad simulation performance.

To avoid this problem, Das et al. introduce two parameters, F_{free1} and F_{free2}. They control at which point an LP is blocked, because it has too few memory left (F_{free1}), and at which (approximated) amount of free memory after garbage collection the LP becomes unblocked again (F_{free2}). Other memory management approaches, e.g. by generating artificial roll-backs, are described in [36, p. 137 -151].

Besides air-traffic control, GTW has also been used in the field of telecommunication. For example, it was used to simulate models in the network description language TeD [76]. GTW is supplemented by the JANE framework [77], which provides interactive simulation functionality. The integration of visualisation components is discussed in [18].

As for APOSTLE, the impact of granularity on the simulation performance was investigated [44]. For the simulation of PNNI, a network protocol for *ATM switches*, different partitioning schemes were evaluated [75]. A load balancing scheme was integrated into GTW, so that an efficient distributed simulation of dynamic models is possible [17]. Jiang et al. describe a framework that enhances the load balancing of GTW on the Grid [50].

IDES

The Infrastructure for Distributed Enterprise Simulation (IDES) is a simulation system developed by Nicol et al. [71] at the Sandia National Laboratories. It is an optimistic simulator written in Java, focusing on portability and the execution in heterogeneous environments.

IDES employs a slightly modified BTB protocol (see description of APOSTLE on page 26) for synchronisation, which is mainly characterised by a different calculation of the local event horizon (LEH). In IDES, all entities executed on one physical processor are regarded as being one LP. Thus, messages between entities on the same processor will be ignored for LEH computation. Instead of considering the time stamps of newly created events, the *received* messages from entities on other processors are used to calculate the LEH. With regard to them, the same synchronisation mechanism is used as in the BTB protocol: When the minimal time stamp of a received message is less than the next event to be processed, the event processing is stopped and the LEH is set to the minimal time stamp of the received messages. Then, the global event horizon (GEH) is calculated as usual, and all processors roll back to it. Now, each processor removes all received messages with time stamps greater than the GEH (because they are unsafe to process), and the same procedure can be repeated.

Due to coalescing all entities on a physical processor and treating them as one LP, so that intra-processor messages are left out, the GEH calculated by this approach may be larger than the GEH calculated by the original BTB simulator. This advantage of a larger synchronisation window is somewhat hampered by the fact that it also increases the cost of state saving and makes roll-backs potentially more complex.

Additionally, the size of the synchronisation window adapts to the available memory. When the amount of free memory is below a certain limit, the processor sets the time stamp of the last processed event as its LEH and stops processing. This corresponds to the memory control mechanisms in SPEEDES and GTW, as Nicol et al. point out, because *"[...] the BTB window is the moral equivalent of GVT."*[71, p. 235].

Nicol et al. also estimated the cost of optimistic and distributed simulation by letting IDES run with and without those features on a single processor. When enabling state saving and synchronisation, the number of processed events per second dropped by approximately 30%. To better predict the performance of IDES, Nicol et al. also developed a performance prediction model [70] (see section 3.3). IDES was used for large-scale interactive simulations of nuclear weapon maintenance [5].

PARSEC

The Parallel Simulation Environment for Complex Systems, PARSEC, was developed at the University of California by Bagrodia et al. [6]. Unlike many other PDES systems presented here (e.g. GTW or IDES), it employs a process-oriented world view instead of an event-oriented one. It is a simulation language based on C and a formerly developed PDES language, Maisie [8].

It consists of three parts: Parsec, the basic PDES system, Pave (Parsec Visual Environment), a user interface for Parsec, and finally Compose, which is a C++ library to facilitate the execution of parallel simulators. Besides that, PARSEC applications use the MPC (message passing kernel for C, [66]). Thus, a PARSEC application is actually a C program that uses some external libraries. This allows to migrate PARSEC applications easily from one platform to another.

For synchronisation, PARSEC supports a broad range from conservative to optimistic protocols. Besides that, the use of different communication layers is possible (e.g. MPI or PVM). Finally, PARSEC incorporates several partitioning algorithms, which vary in their degree of automatism from user-controlled to fully automatic. It is, along with JAMES II, one of the most flexible PDES systems in this overview.

SPEEDES

The Synchronous Parallel Environment for Emulation and Discrete Event Simulation (SPEEDES) was developed by Jeff Steinman [94, 95] at the Jet Propulsion Laboratory. Wieland et al. describe SPEEDES as *"[...] a second generation Time Warp Operating System [...]"*[101, p. 105] that *"[...] benefited greatly from lessons learned during the development of TWOS [Time Warp Operating System]."* [101, p. 105].

This means, SPEEDES aimed at resolving the performance problems that were caused by an overoptimistic execution.

A first solution to this problem was the Breathing Time Bucket (BTB) protocol (as described in the section about APOSTLE, on page 26), which was originally designed for SPEEDES. Besides BTB synchronisation, Steinman also implemented the original Time Warp Algorithm and developed an new approach, Breathing Time Warp (BTW), that can be seen as a mixture of both [95].

Basically, the Breathing Time Warp protocol combines Time Warp and BTB by executing them alternately. Steinman presumes that the probability of an event to be rolled back increases the further it is ahead of GVT. Thus, BTW executes all events that are near the GVT like Time Warp does. But when a processor has processed N events, N being a user-defined parameter, it switches to the BTB phase and will not send out new messages anymore. When all processors have reached their local event horizon, the GVT will be calculated and the protocol starts over again.

As Wieland et al. report in [101], SPEEDES employs a slightly different understanding of LPs and events: Here, LPs are just used as "data objects" to store the state, and they only provide basic functions to access it. In contrast, the main programming logic is situated in the event classes (SPEEDES is written in C++). Each event holds the code to process itself, including the scheduling of new events. Thus, the data objects are not simulation specific and may also be used as database interfaces etc.. The system is thereby divided into classes that contain the model semantics (i.e. the events) and classes that contain or provide the data the model is using.

Wieland et al. evaluated the performance of different SPEEDES configurations with respect to a model of air traffic named DPAT (Detailed Policy Assessment Tool), which consists of a queue network that represents airport runways [101].

The DPAT model was executed using Time Warp, BTB and BTW synchronisation. Additionally, a sequential mode of SPEEDES has been considered. To study the influence of *risk*, which in the context of optimistic synchronisation techniques means the sending of messages although they might have to be rolled back, Wieland et al. introduce two parameters, N_{risk} and N_{opt}. Clearly, Time Warp is a risky protocol, since all messages are sent right away. On the other hand, BTB is clearly a no-risk protocol, since no message is sent that could be rolled back later (recall that all messages are stored on the sending host until the global event horizon is computed).

Now, N_{risk} and N_{opt} can be used to fine-tune the BTW protocol. Each LP will stop its Time Warp phase after processing N_{risk} events, and it will stop its BTB phase after having processed an overall number of N_{opt} events. Of course, if it reaches its local event horizon before it could process N_{opt} events, it will stop as well. If $N_{risk} = 0$ and N_{opt} is set to infinity, BTW is equivalent to BTB. On the other hand, if both N_{risk} and N_{opt} are set to infinity, BTW equates Time Warp. So, the behaviour of the protocol can be controlled seamlessly by varying N_{risk} and N_{opt}[3].

Wieland et al. executed various simulation runs, with astonishing results: Surprisingly, the sequential DPAT simulation outperformed three of the four parallel approaches. Wieland et al. concede that the granularity of the DPAT model is quite low, and that the SPEEDES design seems to be suitable for sequential simulation as well.

However, they also point out another implication: *"The second implication is more foreboding: slow running extant sequential discrete event simulation models probably contain a lot of room for sequential performance improvements. If these performance improvements are made, then the case for running parallel is weakened considerably."* [101, p. 107]. In the case study, only BTW, used with $N_{risk} = 800$ and $N_{opt} = 3000$ (which was the best setup according to Wieland et al.), brought a speedup of 2.1 on 9 processors.

Nevertheless, several load balancing algorithms were integrated into SPEEDES and analysed for their use regarding the DPAT model [102, 103]. Additionally, Steinman developed an interactive version of SPEEDES [93].

[3]It is assumed that $N_{risk} \leq N_{opt}$, otherwise the parameters would not make sense.

SSF

The Scalable Simulation Framework (SSF) is actually not a PDES system, but a specification for which several PDES system implementations exist [21, 89]. The specification describes an API to create processes and events, and specifies interface declarations for C++ and Java. SSF is mainly used to simulate large-scale networks. Basically, SSF can be used to implement sequential as well as parallel simulation systems. It supports a process-oriented world view, which also includes mechanisms for the event-oriented world view.

One PDES implementation, named DaSSF, was developed by Liu et al. [56]. It is written in C++ and will serve as an example of this PDES approach. One of the most interesting aspects described in [56] is the performance prediction of DaSSF, which could be done with an accuracy of approx. 10% (see section 3 for more details). As a reason for performance prediction, Liu et al. argue: *"We assert that the ability to accurate [sic] estimate performance without running a model is crucial to support rational software design, and can greatly boost one's confidence that the effort one puts into model design will yield the desired performance gains."*[56, p. 1].

However, the performance prediction does not take load balancing into account, i.e. it assumes a perfect workload balance. According to Liu et al., this does not influence the prediction strongly, since the load balancing algorithms DaSSF provides are not costly. Clearly, this might not be the case for another kind of model, or another PDES system.

Furthermore, the predicted results are only valid for an infrastructure with similar event queue implementation, computing power, etc., so that Liu et al. summarise: *"We hope at some point to be able to automate much or all of this process; at present, to move to a different architecture is to require that the whole study be done once again by hand."*[56, p. 1].

Unlike the other PDES systems presented here, SSF only allows a conservative synchronisation approach. At first, all processors compute events within the system-wide minimal lookahead. Then, they synchronise using a barrier synchronisation (which only releases the processes when *all* have reached the 'barrier') and send the generated events. Finally, all processes synchronise, again using barrier synchronisation, and repeat generating events within the lookahead.

WARPED

WARPED is an optimistic PDES system developed by Martin et al. [60], in order to *"[...] release a system that is freely available to the research community for analysis of the Time Warp design space."* [60, p. 1].

To motivate their goal, Martin et al. argue that former comparisons between different optimistic synchronisation protocols might be invalid, since *"[...] these investigations are generally conducted in distinct environments with each optimization reimplemented for comparative analysis. Besides the obvious waste of manpower to reimplement Time Warp and its affiliated optimizations, the possibility for a varying quality of the implemented optimizations exists."*[60, p. 1]

WARPED uses MPI as a communication layer. It has been developed in 1995, and was re-designed in 2003 [61]. The new version supports the possibility of runtime configuration and has a more modular architecture. While the first version did not include any partitioning or load balancing mechanisms, Subramanian et al. [96] provide a partitioning framework for the current version.

Although all these efforts aimed at enhancing the comparability of optimistic PDES algorithms, Martin et al. concede: *"'Many groups have been (and are) working on various Time Warp optimizations and reporting results through various publications. Often there is no easy way to make direct comparisons between these optimizations. For example, many global time estimation algorithms for Time Warp have been developed [...]. However, since they have been developed in different systems, it is difficult if not impossible to make meaningful comparisons between them. At best, one must reimplement each optimization within their own system; at worst, a comparison is impossible due to architectural differences or resource limitations."'*[61, p.2].

YADDES

The Yet Another Distributed Discrete Event Simulator (YADDES) is a simulation language developed by Preiss et al. in 1988 [81, 82]. It is based on C and was developed to compare different synchronisation protocols. In [81], the performance differences between optimistic and conservative synchronisation were evaluated, with strongly varying results that emphasized the impact of the simulated model. However, the simulation runs presented in [81] have been conducted on a single computer using parallel processes, so their validity is unclear.

Preis et al. also show how the required transfer time of a message between two LPs can be estimated, although they have to make several very restricting assumptions in order to do so (e.g. computation and communication load are evenly distributed, and all computation and communication can be done in parallel) [81].

Summary

Clearly, not all notable information about the PDES systems surveyed here could be given. Instead, I focused on the synchronisation protocols that have been implemented, since these were described exhaustingly in the corresponding literature. The various synchronisation protocols (BTB, BTW, etc.) were described to give an impression of the diversity and complexity of *one* of the PDES challenges mentioned in section 2.2 (albeit the probably most complex). Some theoretical background on load balancing will be presented in chapter 5. An overview of all aforementioned PDES systems is given in table 2.2 on page 32.

Finally, this survey indicates some major methodological problems:

- *How can algorithms for PDES be compared conveniently?*
  Many PDES systems (e.g. GTW) were designed to gain optimal speedup on a certain infrastructure. As Martin et al. point out in [61], this makes a fair comparison of approaches tedious or impossible (see description of WARPED, page 30). Moreover, the quality of different implementations cannot be guaranteed to be optimal. Consequently, although many of the surveyed PDES systems aimed at providing a comparison platform (e.g. WAPRED and YADDES), and there are many isolated comparisons of PDES algorithms (e.g. [7, 103]), these results may not be valid in general. On top of this, the lack of benchmark data regarding reference models makes a fair comparison virtually impossible.

- *How can we predict the performance of a PDES system?*
  As can be seen in the survey, many researchers contributed new approaches to the field of parallel and distributed simulation. Some of them (e.g. Liu et al. for SSF [56] and Nicol et al. for IDES [70]) also tried to predict the performance of their systems. However, useful formulas could only be found when making very strong assumptions, or leaving out certain aspects that might be important (see section 3.1 for details).

Of course, one could argue that there is no imperative need to predict simulation performance or compare different existing approaches: Instead, trial and error could be used to find the right solution for each simulation problem, even if this is costly. But considering the scientific entitlement of PDES research and basic scientific theory, the published results have to be *falsifiable* and *repeatable*. Indeed, the results *are* falsifiable and repeatable for their *specific* setup, but not necessarily in any other scenario, which obviously complicates their broad application.

The numerous approaches to compare synchronisation protocols etc. show that this view is generally shared within the PDES community, and that a lack of these scientific prerequisites is recognised. Some approaches to overcome these difficulties are discussed in the next chapter.

Name	Implementation	Partitioning	Load Balancing	Synchronisation	Application	Specifics
APOSTLE	C++	-	-	optimistic (BTB)	hardware, health care	granularity control
GTW	C	depending on the application	yes [17, 50]	optimistic, different approaches	network protocols, etc.	originally designed for cache-coherent shared memory processors
IDES	Java	-	-	optimistic (modified BTB)	large-scale networks, interactive simulation	-
James II	Java	yes [29]	under development	optimistic or conservative	agents [39], bioinformatics [25], etc.	currently focusing on efficient simulation of DEVS models
PARSEC	C and C++	different algorithms	-	optimistic, conservative, or mixed	wireless networks, hardware, etc.	-
PDES-MAS	C++	-	yes [73]	optimistic	multi-agent systems	additionally, problems of accessing shared data occur
SPEEDES	C++	yes [102]	yes [103]	optimistic (BTW)	air traffic [101], war games, etc.	different world view: data and event objects
(Da)SSF	C++	-	yes [56]	conservative	large-scale networks [72]	based on SSF specification
WARPED	C++	yes [96]	-	optimistic	queuing networks, hardware	-
YADDES	C	-	-	optimistic, conservative	hardware simulations [81]	-

Table 2.2: Overview: A Selection of Distributed Discrete-Event Simulation Systems

3 Predicting Simulation Performance

This chapter discusses possible solutions for the shortcomings outlined at the end of the last chapter, namely the *comparability* and *predictability* of PDES systems. As the chapter heading suggests, the focus will be on ways to *predict* the performance of PDES systems. However, some approaches also allow a fair comparison of PDES algorithms.

The presented solutions range from analytical to empirical methods. At first, some purely analytical techniques are described, exemplified by a PDES model to analyse the impact of partitioning and synchronisation on scalability [68]. Then, different hybrid approaches, i.e. approaches that use mathematical methods *and* empirical data, are outlined. Afterwards, I give examples of empirical methods, which simulate PDES systems to predict their performance.

Finally, I will argue why the simulation of PDES systems is a good option for predicting and comparing them. Possible problems that could hamper this approach are discussed, as well as additional benefits.

3.1 Analytical Approaches

An interesting approach that highlights both problems and possibilities of analytical performance predictions was presented by Nicol in [68]. Here, I will sketch his attempt to define architectural and algorithmic requirements for a *scalable* PDES system.

Nicol defines a mathematical model which then will be analysed and transformed. His model bases on the typical PDES idea of distributed simulation: The model to be simulated consists of *entities*, which are connected via *channels*. Additionally, he assumes that channels are uni-directional and have a *non-zero* delay $d > 0$, which means that the time stamp of a message sent through a channel has always to be greater than the simulation time of its creation[1].

Now, λ_i is defined as the event rate of an entity i, which is the average number of all events associated with the entity, per unit of simulation time. In this context, association means that the event is *managed* by the entity, so that λ_i refers to the average event queue length of entity i. Correspondingly, μ_k is the average number of messages sent over channel k during one unit of simulation time.

The N processors used by the modelled PDES system are assumed to be homogeneous, i.e. they all have the same computing power, network interface and storage. When partitioning the model, each entity has to be placed on one processor. So, the partition P can be seen as a mapping from the set of all entities into the set of all processors. $\Lambda_j(P)$ is the sum of event rates (λ_i) for all entities that are mapped to processor j by the partition P. This model suffices to define some general 'cost' measures, that actually determine the wallclock time needed to perform certain operations.

Firstly, the cost of processing a single event is defined as $X \cdot log(\Lambda_j(P))$, because Nicol further assumes that each processor has a single event queue. This event queue has the size $\Lambda_j(P)$, and it is assumed that the time to retrieve an event is proportional to $log(\Lambda_j(P))$. Since the event needs to be processed as well, and the mere proportionality between $log(\Lambda_j(P))$ and the time to retrieve the next event does not suffice to define the exact cost of accessing the event queue, $log(\Lambda_j(P))$ is multiplied by a *"problem-dependent constant of proportionality"*[68, p. 5], which is X.

Secondly, the cost of sending an event is defined. The number of messages that are sent from processor i to processor j during one unit of simulation time is called the *communication rate* $c_{i,j}(P)$. $c_{i,j}(P)$ depends on the partition P, because it can be calculated by summing up the μ_k of all channels from entities on processor i to entities on processor j. Moreover, another proportionality constant M and the function

[1] Although this assumption is rather strong, Nicol argues that a model with zero-delay channels can be transformed into one without by aggregating events.

$a(N)$ are multiplied to this term. While M is used the same way as X to 'calibrate' the abstract rates to a physical system and thus reflects the network-related costs of sending a message, $a(N)$ is the function of the architecture size. This is introduced, because the scalability with respect to a growing architecture is one of the issues to be investigated. Therefore, $a(N)$ has to be proportional to the average number of *network hops* a message needs for getting from one processor to another. On top of this, possible contention due to architecture growth has to be modelled in $a(N)$ as well, whereas other contention effects can be expressed by adjusting M.

Thirdly, a processor i that is receiving a message from processor j needs to insert the corresponding event into its own event queue, a cost that again will be proportional to $log(\Lambda_i(P))$. Again, a proportionality constant R will be multiplied with $log(\Lambda_i(P))$, so that the costs can be adjusted to match real-world circumstances.

Using these cost definitions, the total execution cost for processor i during $\delta(P)$, which is defined as a 'natural' simulation time interval for which the costs are determined[2], is defined as

$$\begin{aligned} T_i(P) \leq \delta(P) \cdot (\Lambda_i(P) \cdot X \cdot log(\Lambda_i(P)) \\ + \sum_{j \neq i} (a(N) \cdot M \cdot c_{i,j}(P) + R \cdot \log(\Lambda_i(P)) \cdot c_{j,i}(P))) \\ = \delta(P) R_i(P). \end{aligned} \tag{3.1}$$

As can be seen, $\delta(P)$ is multiplied with the weighted sum of the three aforementioned cost components. Since processor i has to handle $\Lambda_i(P)$ events per time unit, the overall cost for processing events during one unit of time is $\Lambda_i(P) \cdot X \cdot log(\Lambda_i(P))$. The processor will also exchange messages with other processors. This is expressed with the sum over all processors $j \neq i$: processor i will send $c_{i,j}(P)$ messages to another processor j, which costs $a(N) \cdot M \cdot c_{i,j}(P)$ time, and will in turn receive $c_{j,i}(P)$ messages from processor j, which amounts to a cost of $R \cdot log(\Lambda_i(P)) \cdot c_{j,i}(P)$. The overall cost sum per unit of simulation time for processor i is denoted $R_i(P)$.

Further, Nicol defines a 'bottleneck' value $T_{max}(P) = \max_i\{T_i(P)\}$, and correspondingly $R_{max}(P) = \frac{T_{max}(P)}{\delta(P)}$, which is the cost sum for the bottleneck processor. If all parameters (i.e. $a(N)$, X, M, R) are adjusted correctly, $T_{max}(P)$ defines the end time of the entire simulation for the given interval, since all processors are assumed to run concurrently[3].

Now, synchronisation costs are added to the model. The synchronisation costs for one unit of simulation time are defined as $S(P)$, so the costs for an epoch of $\delta(P)$ is $\delta(P) \cdot S(P)$. Hence, the overall costs for simulating $\delta(P)$ units of simulation time are $T_{max}(P) + \delta(P) \cdot S(P)$.

Nicol's approach for testing whether a PDES setup is scalable bases on the notion of *processor utilisation*. In this context, it is the ratio of sequential simulation cost to the cost for using the PDES setup instead. Thus, it is a measure for the overhead induced by using parallel instead of sequential simulation: When then processor utilisation is rather high, i.e. it is only insignificantly smaller than 1 (the theoretical upper bound), there is negligible overhead due to parallel simulation, and one will achieve nearly linear speedup (e.g. a speedup of 4 using 4 processors). In contrast, a low processor utilisation indicates a high overhead. If the utilisation is lower than $\frac{1}{N}$, N being the number of used processors, a parallel simulation will even be slower than a sequential one. Therefore, processor utilisation is a good ratio to evaluate the performance of a PDES setup.

To define the cost for a sequential simulation of the same model, Nicol defines $\Lambda = \sum_i \lambda_i$ as the overall event processing rate of all entities in the model. This can be used to approximate the event processing costs for a single processor. Therefore, a lower bound for processor utilisation $U(P)$ can be defined for a PDES system with N processors:

[2] $\delta(P)$ is only dependent on P for generality purposes, as Nicol states, so that different synchronisation and partitioning mechanisms might be evaluated using different analysis intervals

[3] If this is not entirely the case, this can be expressed by adjusting the parameters.

$$
\begin{aligned}
U(P) &\geq \frac{\delta(P) \cdot \Lambda \cdot X \cdot log(\Lambda)}{N \cdot (T_{max}(P) + \delta(P) \cdot S(P))} \\
&= \frac{\delta(P) \cdot \Lambda \cdot X \cdot log(\Lambda)}{N \cdot (\delta(P) \cdot R_{max}(P) + \delta(P) \cdot S(P))} \\
&= \frac{\Lambda \cdot X \cdot log(\Lambda)}{N \cdot (R_{max}(P) + S(P))}
\end{aligned} \tag{3.2}
$$

To investigate the impact of different parameters on the scalability of a PDES setup, one can now evaluate their impact regarding equation 3.2: if a parameter causes the lower bound to approximate 0, one can say that the PDES setup is *not* scalable regarding this parameter. Hence, Nicol defines a PDES setup to be *scalable* if the expression in 3.2 remains constant or grows when adjusting certain parameters.

Next, he investigates the impact of architecture size and problem size on scalability. To do so, a growth function $g(N)$ is introduced. $g(N)$ specifies the number of model entities on the basis of the number of available processors. Nicol constraints the growth by three assumptions:

1. For every entity i, $\lambda_i = O(\frac{\Lambda^{(N)}}{N})$, with $\Lambda^{(N)} = \sum_{i=0}^{g(N)-1} \lambda_i$
2. For every entity i, the sum of the μ_k on its outgoing channels is $O(\lambda_i)$
3. Every message causes the scheduling of an event for the receiving entity

The first constraint prevents the model from growing in such a way that a single entity has to process so many events that it overloads a processor, even if it is the only entity hosted there. Thus, as Nicol points out, 'serial bottlenecks' for event processing are excluded. The second constraint relates the λ_i, which is constrained by the first assumption, to the number of messages an entity generates. Thus, a serial bottleneck regarding the sending of messages is excluded as well. Finally, the third assumption links the first assumption (bounded event processing rates) to the amount of received messages, so that serial bottlenecks are excluded here as well.

The *Landau notation* (big-oh notation) used for the first two constraints expresses that the assumptions solely describe *upper bounds*. These assumptions are reasonable, because no PDES system would have a chance to scale when a single serial bottleneck (of any kind) is existent.

Then, Nicol proves that every model that satisfies these constraints can be partitioned with a partition P', so that the largest event rate of a *processor* is $O(\frac{\Lambda^{(N)}}{N})$ as well[4]. Consequently, he introduces a constant c_1 so that $\Lambda'_0 \leq c_1 \cdot \frac{\Lambda^{(N)}}{N}$, where Λ'_0 is – without loss of generality – the event rate of the most loaded processor.

Now, given that the model also fulfils the second and the third constraint, which means that incoming and outgoing messages are also bounded by $\Lambda_i(P')$, which in turn is bounded by $c_1 \cdot \frac{\Lambda^{(N)}}{N}$, Nicol provides an upper bound for the cost of network communication known from equation 3.1:

$$
\begin{aligned}
&\sum_{j \neq i}(a(N) \cdot M \cdot c_{i,j}(P') + R \cdot \Lambda_i(P') \cdot c_{j,i}(P'))) \\
&\leq c_1 \cdot \frac{\Lambda^{(N)}}{N} \cdot (a(N) \cdot M \cdot c_2 + R \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) \cdot c_3)
\end{aligned} \tag{3.3}
$$

Similar to c_1, c_2 and c_3 are constants for the second and the third assumption (i.e. bounds on message send and receive costs, respectively) that fulfil the upper bounds given by the Landau notation, for a sufficient large number of processors. Since every $\Lambda_i(P')$ is bounded by $c_1 \cdot \frac{\Lambda^{(N)}}{N}$ (to fulfil the first assumption), and the second assumption has to hold as well, the number of outgoing messages for one

[4]This is rather technical, so the proof is left out here.

processor (which has a finite number of entities) can be bounded by a constant c_2 to $c_2 \cdot c_1 \cdot \frac{\Lambda^{(N)}}{N}$. The same can be considered for the number of incoming messages and c_3.

Now that an upper bound for each part of the maximal weighted sum of cost, $R_{max}(P')$, is known, this can be combined to:

$$\begin{aligned} R_{max}(P') \leq X \cdot c_1 \cdot \frac{\Lambda^{(N)}}{N} \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) \\ +c_1 \cdot \frac{\Lambda^{(N)}}{N} \cdot (a(N) \cdot M \cdot c_2 + R \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) \cdot c_3) \end{aligned} \tag{3.4}$$

Clearly, this inequality can be deducted from equation 3.1 by substituting $\Lambda_i(P')$ by its upper bound $c_1 \cdot \frac{\Lambda^{(N)}}{N}$, etc.. This enables the expression of a growth-dependent lower bound for the processor utilisation, by inserting the cost bounds into inequality 3.2:

$$\begin{aligned} U(P) &\geq \frac{\Lambda^N \cdot X \cdot log(\Lambda^N)}{N \cdot (X \cdot c_1 \cdot \frac{\Lambda^{(N)}}{N} \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) + c_1 \cdot \frac{\Lambda^{(N)}}{N} \cdot (a(N) \cdot M \cdot c_2 + R \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) \cdot c_3) + S(P'))} \\ &= \frac{\Lambda^N \cdot X \cdot log(\Lambda^N)}{\Lambda^{(N)} \cdot (X \cdot c_1 \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) + c_1 \cdot (a(N) \cdot M \cdot c_2 + R \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) \cdot c_3) + \frac{N \cdot S(P')}{\Lambda^{(N)}})} \\ &= \frac{X \cdot log(\Lambda^N)}{X \cdot c_1 \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) + c_1 \cdot (a(N) \cdot M \cdot c_2 + R \cdot \log(c_1 \cdot \frac{\Lambda^{(N)}}{N}) \cdot c_3) + \frac{N \cdot S(P')}{\Lambda^{(N)}}} \end{aligned} \tag{3.5}$$

This formula can now be analysed. For example, it is clear that the given bound will *not* approximate zero when *only* X is increased. So, if the granularity of our simulation is increased by making the processing of an event more complex in the average case, the lower bound for processor utilisation will *never* approximate zero. This implicates, that *every* PDES system scales with this kind of problem growth. However, as Nicol states, it may be that the synchronisation costs ($S(P')$) rise as well, for example because state saving or roll-back costs increase when synchronising optimistically.

Nevertheless, as Nicol further analyses the deducted formula, he finds some interesting relations. For example, he is able to prove the scalability of a simple conservative synchronisation protocol for a given architecture. The architecture assumption highlights the importance of efficient network topologies, because this could only be proven for a *hypercube*. The network diameter of a hypercube is very small, which makes $a(N)$ proportional to $\log(N)$.

Furthermore, Nicol describes the trade-off between synchronisation and load imbalance when the scalable synchronisation protocol is used. The protocol basically determines the minimal delay d (which is actually the look-ahead) for all channels between *different* processors, and then executes all events within a time window d. The larger the minimal delay of inter-processor channels, the larger is the synchronisation window that is used, and the smaller are the costs of synchronisation.

Now, the partitioning algorithm could at first merge all entities that have a channel connection with a delay smaller than a given value d', and then partition the remaining entities. This would guarantee a synchronisation window size of at least d', but the partly merged entities may be harder to map evenly on all processors. Therefore, Nicol proposes a binary search algorithm to find the optimal relation of synchronisation and imbalance.

All in all, Nicol's approach is quite impressing: It demonstrates that mathematical modelling of PDES systems is very well possible, and that rather general and broadly applicable conclusions can be deducted. Having said that, it is evident that a thorough analysis of such a complex formula, with such a number of adjustable parameters (which even may depend on each other), is anything but easy.

Even for this general approach, several strong assumptions have to be true, for example the execution of the PDES on a set of *homogeneous* processors, non-zero channel delays, and that the cost of accessing an event queue is proportional to $\log(N)$. If one of these assumptions does *not* hold, the analysis has to be repeated. Since there are event queues whose access costs are $O(1)$ under certain circumstances, Nicol points out that *"Reduction of the event list cost is a sort of good news and bad news deal."*[68, p. 11]. On top of this, many of the used parameters are very hard to find out for existing PDES systems. For example, Nicol admits that *"For most synchronization protocols, quantifying $S(P)$ is extremely difficult."*[68, p. 5], and that *"Furthermore, synchonization behaviour is frequently complicated, which makes it very difficult to analytically prove anything about performance executing large models on large machines."*[68, p. 4].

Another restriction is the focus on *average* rates, so that the possibly dynamic behaviour of the model is not modelled explicitly. Consequently, additional costs, for example the cost of load balancing, are not modelled explicitly as well, and thus cannot be analysed directly.

These examples illustrate the biggest shortcomings of the analytical prediction approach: The results, although being quite general, cannot be applied easily to real-world situations. Additionally, the mathematical models tend to be quite complex, and often base on rather strong assumptions. However, these assumptions have to be made, otherwise a mathematical model would not be tractable at all. So, the mathematical prediction of simulation performance is rather laborious, very sensitive to preliminary assumptions, and the results are often hard to apply. Owing to these difficulties, Nicol concludes that *"[...] it becomes increasingly critical to establish whether PDES technology will scale up with increasing problem size and architecture. In this paper, we address the problem in a general setting, provide a resounding conclusion:* **maybe.**"[68, p. 11]. Nevertheless, mathematical models have the strong advantage of being able to show theoretical upper and lower bounds *in general*, so that limitations in principle can be detected.

There are several other analytical approaches, one of them was presented by Gupta et al. [43], who predict the performance of time warp. In contrast to Nicol's approach, they do not search for a theoretical lower bound that can be analysed, but try to predict several performance measurements (e.g. speedup, number of roll-backs, etc.) for a time warp simulation. The resulting equation system was solved numerically, and the results were compared to real-world measurements. The comparison showed that the speedup prediction was rather good, only overestimating the speedup by at most 10%. But, again, the model implies several strong assumptions: For example, the analysis based on the fact that communication takes *no* time at all. While it is clear that inter-processor communication on the shared memory processor that was used to generate the real-world data might be negligible, this is surely not the case when using other network topologies. Actually, Nicol shows in [68] that network communication overhead *does* matter. Furthermore, other aspects of the problem, like dynamic model behaviour, load balancing, or granularity were left out. The equation system and its deduction were nevertheless quite complex and sophisticated.

Altogether, analytical approaches may not be the best way to predict the performance of specific PDES systems. The next section presents some approaches that combine analytical considerations and real-world data to generate predictions more easily. Finally, the aforementioned examples indicate that it is generally quite hard to predict PDES performance. Some reasons for that are suggested in section 3.4.

3.2 Semi-Analytical Approaches

3.2.1 Benchmark Analysis

Unlike purely analytical techniques, which focus on the analysis of rather general PDES system models, one could also enrich a mathematical method with benchmark data from existing PDES systems. This approach was used by Liu et al. to predict the performance of the DaSSF system, as already described in section 2.3.3 [56].

Basically, Liu et al. investigated execution times for low-level operations that had to be performed quite often when running a simulation with DaSSF. Then, the occurrence of these operations was calculated by a straightforward mathematical model. By putting the measured execution times of the operations into the mathematical model, the execution time could be predicted. The prediction quality was rather good for most predictions (approx. 10% deviation), but in some cases it varied by as much as 30% from the real-world execution time.

To gather useful benchmark data, Liu et al. identified several operations that are essential for the performance of DaSSF. Since they implemented an own threading mechanism, the speed of switching the context (i.e. switching the process currently executed) had to be measured. Different experiments were designed to examine the average execution time for different blocking functions. Moreover, the costs of creating process and event objects were evaluated. Finally, they quantified the costs of accessing their event list implementation and of possibly expensive operations that are specific to DaSSF.

Since some measurements depend on external factors, e.g. the performance of an event list usually depends on the number of events it handles, these had to be investigated by performing test *series*. Although this work is very laborious, Liu et al. point out that this would have to be done *anyway*, because this is the only way of determining the parameters that are necessary for *any* prediction. From this point of view, one could regard this type of analysis as an extension of mathematical approaches by integrating benchmark results, so it is named "benchmark analysis" here.

Nevertheless, measuring the exact execution time of single operations or features is quite challenging. Different benchmark models were needed, and these had to be set up carefully, so that accurate measuring was possible. Moreover, Liu et al. allude to *"The Heisenberg principle of performance monitoring (you cannot observe a program without affecting its behaviour) [...]"*[p. 156][56], which can actually be seen as a limitation of observation accuracy in principle.

In general, a benchmark analysis is able to accurately predict the performance of a concrete PDES system using a *specific* setup and configuration, but is otherwise hampered by the same shortcomings as purely mathematical approaches (see section 3.1).

Moreover, the PDES system for which the performance is predicted has to be *existent*, otherwise no benchmark values could be obtained. Although this sounds like a common presumption, the prediction of PDES systems might also be useful to evaluate different simulation setups *beforehand*. The next section presents an approach that fulfils this requirement.

3.2.2 Trace-based Performance Prediction

Another way of predicting parallel performance would be to execute a model sequentially, and to predict how a parallel execution *would* have performed in the same situation. To do so, one needs information about all events that have been generated: *When* did *which* model entity send an event to *which* other entity, *how* long took its processing, and *which* former event caused this behaviour? A list containing this data can be compiled, e.g. sorted by non-decreasing time stamp order of the described events. This list, which is the complete definition of the model behaviour for the observed simulation run, is called a *trace*.

The basic idea is now to analyse these traces and to calculate how much time its execution using a parallel simulation would have taken. As an example, the performance prediction tool developed by Juhasz et al. is further detailed [51, 52].

Juhasz et al. use the so-called *post-mortem* analysis, which means that they let a sequential simulator generate a trace at first, and analyse the trace *after* the sequential simulation has finished. Opposed to this, one could also integrate an on-line trace analyser into the sequential simulator, but the post-mortem analysis is advantageous. For example, it allows to compare several synchronisation protocols after running the sequential simulation just once.

To calculate the predicted speedup, the generated trace is read and transformed into a directed graph. Each event is represented as a node. Edges are generated between each event and the events it caused. Thus, a directed, acyclic graph (DAG) is created, which is named *event precedence graph*. Each node is associated with the time that was needed to process the corresponding event.

Now, the *critical path* of the simulation run can be identified. In this context, it is the path with the maximal processing time, which is defined as the sum of processing times of all contained nodes. The processing time of the critical path is quite interesting, because it can be seen as an upper bound for achievable speedup and gives an idea of the model's inherent parallelism[5].

A sample event precedence graph and its critical path are depicted in figure 3.1. The critical path can be considered as the costliest chain of events that cause each other, and thus *cannot* be executed in parallel.

[5]It can be shown that optimistic simulations could, *theoretically*, outperform this upper bound under ideal circumstances.

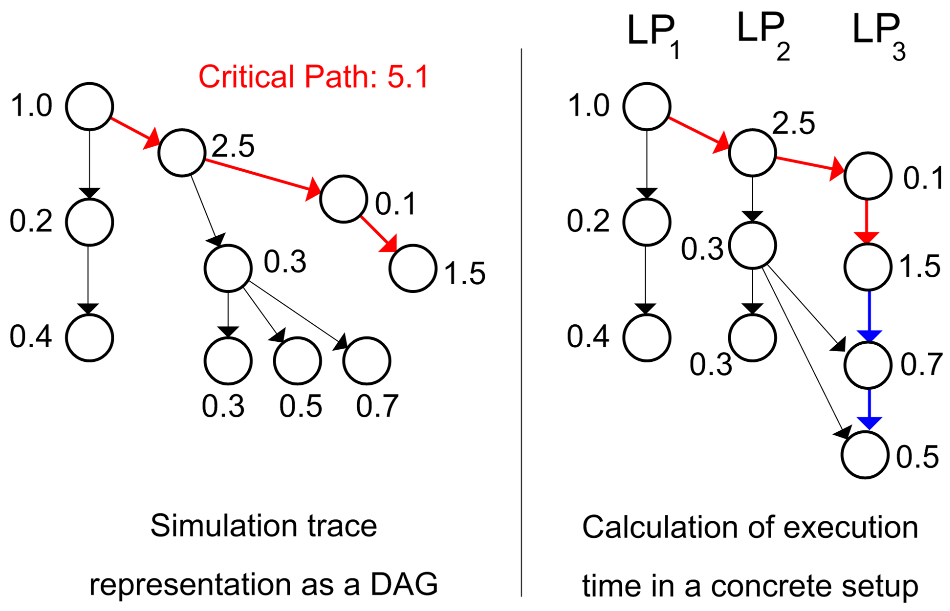


Figure 3.1: Event Precedence Graph. All processing cost labels could, for example, be regarded as execution time in seconds. The red edges define the critical path of the overall simulation run. However, concrete setups as depicted on the right might result in longer execution times, because one LP (in this case, LP_3) might have to process additional events. Sequential processing dependencies can be expressed by additional edges in the precedence graph. They have been coloured blue in this example.

In a sequential simulation, all events are simulated one after another. So, the optimal speedup, assuming that all events which do not belong to the critical path can be simulated in parallel without any overhead, is the ratio of the overall sequential time to the time for all events on the critical path.

Alternative parallel PDES setups can now be considered by mapping the model entities to a set of p imaginary processors. Each event in the event precedence graph will be assigned to the imaginary processor that hosts the entity which processed it. The events that have been assigned to one processor are sorted in increasing simulation time order, and additional edges are added between consecutively processed events (see figure 3.1). Using the new graph, the performance of an optimal PDES execution on p processors can be predicted by identifying the critical path again.

By considering the edges between events on different processors, communication costs can be integrated into the prediction method. This is a very flexible approach, as Juhasz et al. show. They propose a mathematical model for communication cost:

$$T_{comm} = T_s + k \cdot T_w + r \cdot T_h. \tag{3.6}$$

In this model, T_s is the message start-up time, i.e. the time to generate a message and sending it away. Clearly, the transfer time also depends on the length of the message, k, and the number of intermediate network nodes until it reaches its destination, denoted by r. T_w and T_h are constants that represent the additional per-word and per-hop transfer time within the used architecture, so that the overall communication time T_{comm} can be calculated. This allows the prediction tool by Juhasz et al. to immediately calculate the highest possible speedup for the architecture parameters defined by the user.

The characteristics of different synchronisation protocols are emulated by adding additional edges which represent synchronisation messages. Since the predicted wallclock time at which events are calculated may change when doing so, the events may need to be adjusted on their processor timelines. In this way, different synchronisation approaches can be compared. In [97], Teo et al. present an extended version to predict the performance of optimistic synchronisation protocols as well.

All in all, the trace-based prediction approach is very powerful: different synchronisation algorithms can be compared, and their execution can be assessed under a variety of setups. The integration of prediction methods for combinations of different network architectures, processor capacities, and partitioning methods seems straightforward. Juhasz et al. point out that their tool *"[...] "simulates" the parallel execution from runtime trace information generated by a sequential simulation run."*[52, p. 338].

However, there are also some disadvantages: Firstly, the *post-mortem* analysis, while having some advantages, makes it time consuming to measure the impact of certain *model* characteristics, e.g. granularity or dynamic behaviour. Using this way of performance prediction, two steps are needed for every measurement: At first, the trace has to be generated, and then it has to be analysed.

Further problems may occur when intricate mechanisms are evaluated, for example load balancing algorithms or sophisticated synchronisation protocols. Although their impact can be modelled by the analyser in principle, the validation of this prediction is quite hard: How to find a semantic error in an algorithm that simply re-labels some edges to account for certain circumstances? This problem requires thorough and laborious testing.

Besides that, the efficiency and overall performance of the tool is unclear. Unfortunately, there are no detailed performance results or comparisons between predicted and real-world performance to be found in [51, 52, 97]. And despite the fact that there are efficient algorithms to detect critical paths in DAGs [63], it is not clear how well this approach scales. Actually, Liu et al. argue in [56] that the lack of scalability is one of the major drawbacks of trace-based approaches. However, the use of simulation methods to predict PDES performance seems quite promising. The next section presents approaches that use these techniques more consistently.

3.3 Empirical Approaches

As can be seen in section 3.1, mathematical analysis of PDES systems is quite difficult. One way to tackle problems too complicated for mathematical analysis is, obviously, to use modelling and simulation methods.

Perumalla et al. describe such a prediction system in [78]. They attempt to "virtualise" a PDES system, by which they mean that it is simulated. To do so, they developed a framework for the parallel and distributed simulation of PDES systems, consequently named $PDES^2$.

Although the application of a method on itself might seem odd, Perumalla et al. point out that virtualisation is a feasible prediction method for the performance of a PDES system on a supercomputer. The models Perumalla et al. want to deal with, e.g. of the Earth's magnetosphere, are quite large. Hence, they are simulated on a supercomputer. Since supercomputer time is expensive, they propose to use the virtualised PDES system for testing, debugging, and tuning purposes, especially with respect to different synchronisation and load balancing schemes.

Perumalla et al. also identify two additional notions of time for a simulation of a simulation[6], namely the simulation time and the wallclock time of the simulated simulation. However, the wallclock time of the simulated simulation is equivalent to the actual simulation time, so that four notions of time can be distinguished[7].

As an input, characteristical traces for sequential simulation runs have been replicated. The results presented in [78] are remarkable in that they accurately predict the simulation performance for a virtualised simulation on over thousand CPUs. However, some predictions deviated strongly from the actual performance, for which Perumalla et al. hold "[...] platform artifacts (such as operating system schedulers)"[78, p. 362] responsible. Additional to the speedup prediction, PDES^2 also logs further benchmarking information, for example the time that was spent for the synchronisation protocol. This allows to make out which factors have the biggest impact on efficient execution.

[6]See page 9 for a description of the basic notions of time.
[7]This will be detailed in section 4.1.

All in all, this approach seems to be the most powerful of all approaches outlined here, because it can be used as a solution for very different needs: Using simulation for performance prediction, one can easily test and debug existent code, evaluate new approaches not implemented yet, or fine-tune the parameters of an algorithm.

However, the distributed execution of the virtualised simulation may bring some problems: When using PDES^2 to test new approaches, it is likely that one also needs a good support for debugging. Although it is generally possible to debug a system simulated by PDES^2, debugging of parallel and distributed executed code is *not* trivial and may be very tedious, as it is described in [12]. On the other hand, this is still clearly better than allowing 'false positives', which is a shortcoming of other approaches, like trace-based prediction.

Another problem when testing new approaches might be that *all* of the simulation system has to be implemented as a PDES^2 model. This is due to the fact that the simulation model is simulated *in parallel* and *distributedly*. So to say, it is very hard to "cheat" by only modelling the simulation parts of interest. For example, one might want to have a publicly available GVT variable instead of a complete synchronisation protocol implementation when just studying different load balancing algorithms.

Apart from this approach, Nicol et al. also used a time-stepped simulation to calculate their mathematical model of IDES performance [70]. They used a discrete-event simulation because their model could not be calculated easily by hand. The IDES prediction model was used to estimate the differences between the original BTB protocol and some modifications which were later used by IDES (see IDES description on page 28 for details).

Even though the IDES performance model leaves out many important factors, such as load balancing and varying network delay, Nicol et al. are able to show that the performance of both synchronisation methods hugely depends on the computer hardware, network architecture, and the number of modelled entities. Analysing the obtained results, they conclude that *"[...] there is a "cross-over" point where pre-sending begins to be advantageous. We see that good performance is possible, but that relatively poor performance is also possible."*[70, p. 43]. In this case, they identify the granularity to be the factor that defines the cross-over point.

Altogether, simulation approaches seem to be a very promising and "natural" way to predict simulation performance. Besides yielding rather accurate predictions, simulation of PDES systems could be used for many other problems in context of PDES system development. The next section elaborates on further reasons for using this prediction technique.

3.4 Why Simulate Simulations?

Before working out the usage of the simulation approach to predict load balancing performance, it is important to discuss the reasons which cause the simulation approach to be more feasible than analytical ones.

As described in section 3.1, mathematical models use very strong assumptions in order to make the models mathematically tractable. In contrast, PDES techniques are often applied to analyse systems for which mathematical models are unknown or intractable unless incorporating unrealistic assumptions. This is not the case for all modelling and simulation techniques, for example, continuous models are usually described by differential equations, which are essentially a mathematical way of modelling.

Usually, the systems simulated by PDES systems, mathematically intractable and hard to predict, are described as being *complex*. As for all terms that are broadly used, there are dozens of definitions for complexity. Common definitions are e.g. *"Consisting of two or more interconnected parts"*, and *"Intricate; complicated"* [1, p. 181].

Opposed to these definitions, Edmonds argues in [27] that neither size nor mere difficulties in understanding are sufficient conditions for complexity. Instead, he suggests that complexity is a *"[...] property of a language expression which makes it difficult to formulate its overall behaviour, even when given almost complete information about its atomic components and their inter-relations."*[27, p. 5]. Of course, this perspective on complexity is very specific and not even shared by all computer scientists. For example, theoretical computer science associates complexity with the Landau notation.

Interestingly, Edmonds' definition suits very well to the performance prediction of PDES systems in general: Although the developers of PDES systems have *"almost complete information about its atomic components and their inter-relations"*, they are not able to *"formulate its overall behaviour"*. So, a reason for the inability of mathematical models to accurately predict PDES system performance, without assuming unrealistic presumptions, might be that PDES systems can be complex systems by themselves? One could see mathematical expressions as a language, and the formulation of "overall behaviour" would clearly allow an accurate performance prediction.

Another indication that PDES systems might be complex can be deducted from their use to investigate *complex* systems. Firstly, we can assume that models of complex systems are also complex to a certain degree. Otherwise, one could, according to Edmonds' definition, easily formulate their overall behaviour with explicit mathematical expressions and hence would not need to simulate the model at all.

For executing a PDES, a model's entities are distributed over available LPs, so that each LP hosts a set of entities. Since the model is *complex*, it is not possible to predict the overall behaviour of all entities. The unpredictable behaviour also includes the communication between the entities and their requirements regarding computation power. Hence, an LPs' *set* of entities and its computation and communication behaviour could be unpredictable as well.

Since we have knowledge about the model to be executed and the mechanisms that are used for its execution, we have almost complete information of the atomic components that belong to the possibly complex PDES system. But, despite the availability of such knowledge, the analytical prediction of PDES systems is not possible for *arbitrary* models to this day. This consideration is summarised in figure 3.2.

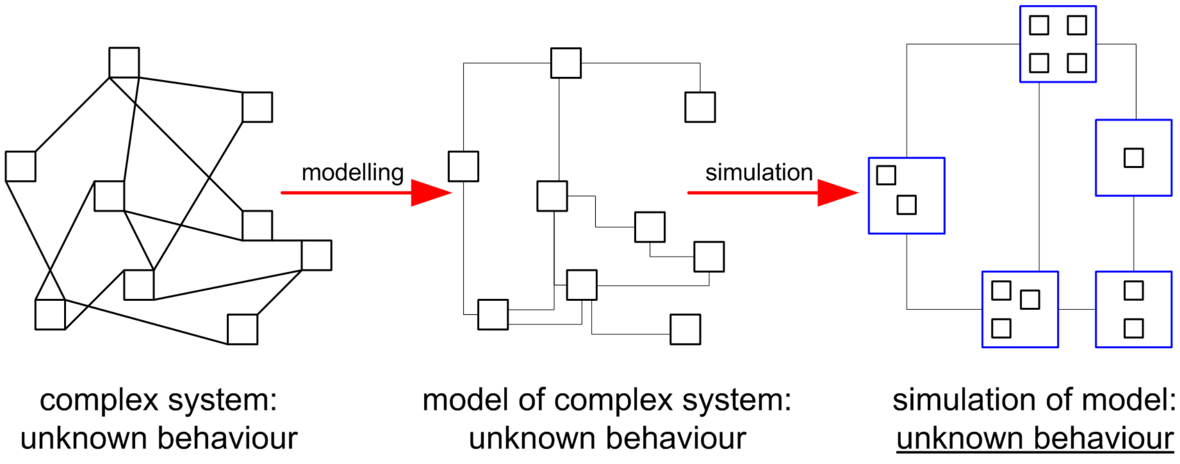


Figure 3.2: How Complex Systems hamper Performance Prediction: When complex systems are analysed, their model's behaviour is hard to predict *as well*. Consequently, the behaviour of the LPs that execute the model is also hard to predict.

The outlined train of thoughts illustrates the impact of model characteristics on PDES system behaviour. Consequently, the complexity of a PDES system depends on the complexity of the simulated model. This would explain why the models that were considered for (semi-)analytical approaches are relatively simple (e.g. Liu et al. used a slightly modified *PHOLD model* [56]).

Finally, the possible complexity of PDES systems implies that the application of modelling and simulation techniques, i.e. the simulation of PDES systems, is *indeed* appropriate and feasible. As a matter of fact, the analysis of complex systems is one of the central application domains for modelling and simulation.

Now that the impact of model characteristics on PDES system performance was illustrated, another challenge for *general-purpose* PDES systems is obvious: How to guarantee an efficient execution of an *arbitrary* model?

Often, algorithms used in PDES systems are only efficient for a *subset* of all possible models. For example, Fujimoto concludes after describing a modification for the Time Warp protocol: *"Depending on the application, performance of Time Warp using lazy cancellation can be better or worse compared to aggressive cancellation."*[36, p. 132]. Due to the diversity of benchmarking models, this also hampers the *comparability* of PDES systems. So, it seems as if the problems of predictability and comparability are somehow inter-related.

To overcome the dependency of PDES algorithms on model characteristics, different *adaptive* mechanisms have been introduced, for example to adapt the optimism of the synchronisation protocol or the behaviour of the partitioning mechanism to the model [23, 14]. This leads to an increasing complexity of PDES systems.

Besides that, many algorithms can be fine-tuned by adjusting certain parameters, such as N_{risk} and N_{opt} in the BTW synchronisation protocol (see SPEEDES description on page 28).

The set of all meaningful combinations of PDES algorithms, tuned by any meaningful combination of parameters and executed on any meaningful hardware setup, can be regarded as a huge n-dimensional space S, which will be referred to as *setup space* in the following. Additionally, we can define a set of all possible models, M, and the *performance function* $f_{perf} : S \times M \rightarrow \Re$, that maps each execution, where a model $m \in M$ is simulated by a PDES setup $s \in S$, to its execution time. The set $S \times M$ is named *simulation space*, since it contains all possible simulation runs.

To develop a *new* algorithm and predict its performance, one has to explore regions of this parameter space. Thus, prediction techniques could be regarded as methods to approximate f_{perf}. From this perspective, analytical methods are used to identify bounds of f_{perf}, whereas semi-analytical and empirical techniques try to estimate f_{perf} as good as possible. Correspondingly, any benchmarking and testing of a simulation system is the *exact* determination of f_{perf}, for a certain setup s and model m.

From this it follows that the aforementioned *cross-over points* Nicol observed in [70] have to be quite common, since the value of f_{perf} may vary along each dimension (see fig. 3.3). So, cross-over points indicate the region, i.e. a subset of $S \times M$, for which one algorithm outperforms the other. Complete knowledge of these cross-over points would allow to select the optimal setup for each model.

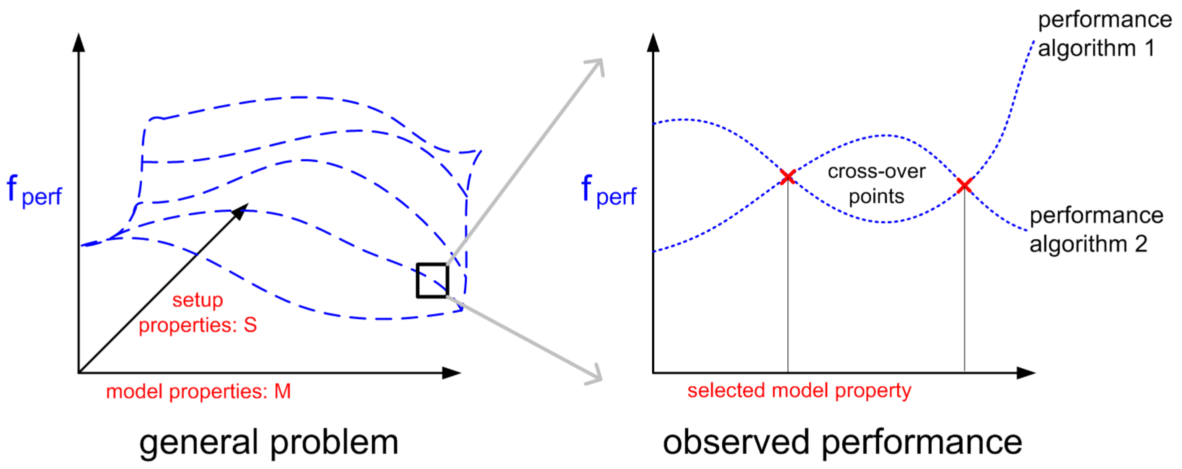


Figure 3.3: Illustration of f_{perf}. Real performance measurements usually cover only small regions of the simulation space, as can be seen in the right plot.

Finally, as a last argument that PDES systems can be seen as complex systems, one could examine what other sciences concerned with systems regard as complexity and adaptivity. Schuster describes complex and adaptive systems from a physicist's perspective as *"[...] made up of elements or agents such as genes,* *neurons or players in a game which interact with each other and with their environment in a nonlinear fashion. Therefore, from the physical point of view, CAS [complex and adaptive systems] are unconventional analog computers which can be described within the framework of nonlinear dynamics."*[88, p. 7]. As examples, Schuster names, among others, the "Prebiotic Evolution" (modelled by chemical networks) and the "Social Evolution" (modelled by agents playing games).

Evidently, a PDES system is by itself *not* an "unconventional analog computer", so PDES systems do not fit into this definition of complex systems. However, the point is that PDES systems *execute* models of complex systems (e.g. the simulation of chemical processes and games played by agents), so models of such systems cause the PDES system behaviour to be unpredictable and complex to a certain degree, too. However, there are some concepts of complexity that do not even roughly agree with its notion used in this work. For example, Rosen states that *"[...] organisms possess noncomputable, unformalizable models. Such systems are what I call complex."*[87, p. 4]. This would mean that none of today's computer systems could be complex by definition, since they are inorganic. Rosen's definition also includes software that runs on such systems. Obviously, this view on complexity is *not* coherent with the notion of complexity as described before.

Nevertheless, it is clear why the simulation of PDES systems might be a good way to estimate PDES performance. The next section will outline how a simulation simulator could facilitate PDES algorithm development.

3.5 Towards a Proof of Concept

It is often highlighted how complicated and time-consuming the development of new algorithms for PDES systems is. Nicol et al. state that *"A bevy of optimistic synchronization protocols exist, all complex and all challenging to program correctly and efficiently."*[71, p. 234]. Alike, Liu et al. argue that *"The results of 20 years of research in parallel simulation reveal it to be a highly complex endeavour, with performance results very much dependent on implementation details and model characteristics."*[56, p. 1].

An accurate and fast estimation of f_{perf} would help to overcome these problems, because the performance of a setup containing a new algorithm could be predicted. This would allow a testing of the algorithm without having to actually provide the setup in which the developed algorithm should eventually run. This advantage was the motivation for Perumalla et al. to virtualise a PDES system using PDES^2 (see section 3.3).

Simulating a PDES in order to estimate f_{perf} brings in fact several additional advantages. When simulating a PDES algorithm at first, the simulation may identify design problems very early. Moreover, a model of a PDES algorithm should be smaller, more concise and not dependent on low-level functionality. This further facilitates problem detection. On top of this, different algorithmic alternatives and their impact on the overall performance may be evaluated. A simulation simulator may also be used for other purposes, which are beyond the scope of this thesis. Some of them are outlined in section 7.3. Now, it is more important to prove the points that were made in this section.

To do so, a load balancing algorithm for the abstract PDEVS simulator of James II will be developed by using a simulation simulator to predict its performance beforehand. The next chapter describes the implementation of such a simulation simulator, SimSim, with which the PDEVS execution of James II will be simulated.

Afterwards, the usefulness of the approach to develop PDES algorithms can be assessed. Since prior modelling of an algorithm (and all relevant parts of the PDES system it is executed in) could bear significant additional work, this will have to be weighted carefully against the possible advantages. Furthermore, it is not yet clear how fast the predictions can be generated and how accurate they are. In fact, this is the crucial point of the proposed approach.

4 SimSim: A Simulation Simulator

This chapter describes the development of a general purpose simulator for PDES systems, SimSim. It is based on a tool to predict the performance of the PDES-MAS simulation system when using different routing algorithms (see section 2.3.2). The original tool was developed at the University of Birmingham and specifically tailored for the needs of PDES-MAS [30].

Hence, the whole tool had to be re-designed and enhanced. At first, section 4.1 clarifies some terminological issues. In section 4.2, general requirements for a simulation simulator are worked out. This is supplemented by section 4.3, which discusses the shortcomings of the PDES-MAS simulation tool. Section 4.4 introduces a generic model for PDES systems, and section 4.5 outlines basic architectural decisions, which are resulting from the aforementioned shortcomings, requirements, and the generic model. Finally, section 4.6 gives some implementation details, thereby focusing on certain problematic issues.

4.1 Additional Terminology

The accurate description of a simulation simulator demands specific terminology, so that new concepts will not be confused with common modelling and simulation terms. Evidently, the simulation of a simulation introduces a new layer to the common division of simulation programs into parts that define a model and parts that simulate it.

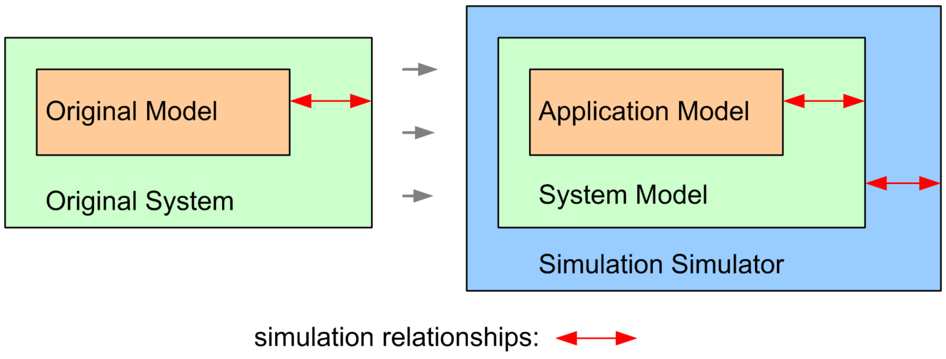


Figure 4.1: Nested Modelling of PDES Systems: When modelling a PDES system (left), an additional simulation layer is added (right), so that the original terms have to be re-defined.

For the sake of clarity, the PDES system to be modelled is called *original (PDES) system*, and the model it executes is called *original model*. In contrast, the model simulated by the *modelled* PDES system is named *application model*, whereas the model of the PDES system itself is called *system model*. The structure of a simulation's simulation and the denotation of the different parts is depicted in figure 4.1.

Moreover, as already mentioned in section 3.3, there are two additional notions of time when simulating a simulation, namely simulation time and wallclock time of the additional simulation layer. Since the physical time is irrelevant in this context, and the simulation time of the simulator equals the wallclock time of the original system, it is sufficient to distinguish between three different notions of time: *application model time*, *simulated wallclock time*, and wallclock time (see fig. 4.2).

As before, wallclock time means real-world time, which is now the runtime of the simulation simulator. The simulation time of the simulator is now called simulated wallclock time, to highlight the fact that it represents the wallclock time of the original PDES system. Finally, the application model time denotes the simulation time of the original system.

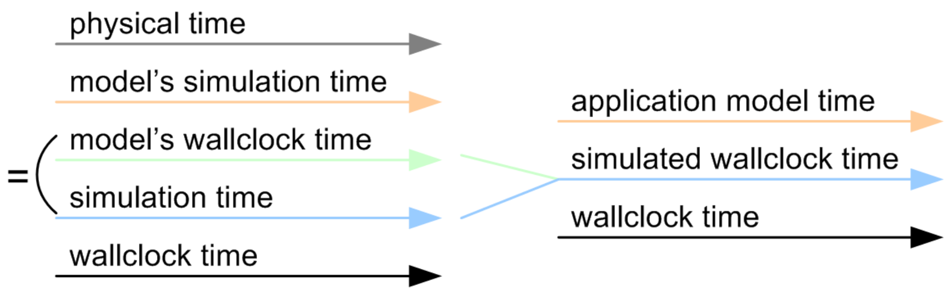


Figure 4.2: Time Notions for Simulation of PDES Systems: From five notions of time that were identified by Perumalla et al., only three notions need to be distinguished in this context. The colour of each arrow corresponds to the part of the system where this notion of time is used (see figure 4.1).

While this differentiation of timelines is sufficient here, it has to be stressed that these simplifications might not be useful in other cases. For one, the simulated wallclock time does *not* need to be exactly equivalent to the wallclock time of the modelled PDES system. When simulating a simulation run from 4:00 to 5:00 P.M., the simulated wallclock time could run from 0 minutes to 60 minutes instead. So, the wallclock time of the simulated PDES system can be regarded as a physical time, which in turn is represented by the simulation time of the overall system: the simulated wallclock time.

Furthermore, the application model time might also differ from the simulation time of the original model. This is the case when original model and application model are *not* equivalent. For example, creating a more abstract application model that only approximates the original model's behaviour may be useful to save development time.

4.2 Requirements

Above all, a general purpose PDES system simulator has to support the simulation of *different* PDES systems for *general* purposes. For example, requirements for developing load balancing algorithms may differ strongly from those for developing synchronisation protocols. Obviously, the employed models may vary with respect to different purposes of PDES system simulation. This results in unpredictable additional requirements regarding parameters and measurements.

Although one might be able to define, for example, all possible performance measurements (like speedup, processed events per second, processor utilisation, etc.) beforehand, this solution has several drawbacks: Evidently, it will be quite time-consuming to list *all* measurements for *all* kinds of algorithms that might be evaluated (granularity control, routing, synchronisation, etc.). Furthermore, users of the simulation tool will still have to implement code to *determine* these measurements. For other demands, like the definition of parameters to control application models, even a prior definition is impossible. Therefore, a general purpose PDES system simulator has to provide means for extending its functionality and implementing application-specific adjustments.

This can be done by defining an interface for *plug-ins*, which are, as the name suggests, software components that can be easily plugged into a program, in order to enhance its functionality. A plug-in may contain several system and application models. Consequently, it has to define all parameters that can be adjusted, as well as all measurements that can be observed.

Now, a PDES system simulator can assist the developer of a new algorithm in at least two ways: On one hand, it can be used for debugging an algorithm, or observing its behaviour under certain circumstances. On the other hand, the tool can be used to observe algorithms under a *variety* of circumstances, and thus predicting their performance for a certain region of the simulation space. The tool has to support *both* possibilities by providing appropriate user interfaces and functionality.

To support debugging and scrutinising of an algorithm's behaviour in a single run, one might need to adjust the speed of the simulation or pause it. Moreover, a visualisation of the algorithm's behaviour could facilitate this task, and the developer might even want to adjust certain settings *interactively*. In contrast, visualisation is expensive and useless when it comes to simulating hundreds or even thousands of simulation runs, as it could be the case when an algorithm's performance needs to be predicted for various simulation setups and model properties.

An efficient usage requires additional functionality: Firstly, flexible and stable methods to obtain the data for various measurements, and secondly a mechanism to run a large number of simulation runs automatically, so that f_{perf} can be predicted for *regions* of the simulation space. The tool should be able to simulate a *batch* of simulation space elements, and to save *various* measurements (not just f_{perf}) for later analysis.

The selection of measurements and the configuration of batch mode parameters is essential for the analysis of the obtained results. Moreover, system and application model might also be subject to parameter adjustments, which has an impact on the results as well. Since reproducibility is crucial for any kind of experiment, simulation experiments included, these parameters have to be store- and loadable. Moreover, it is desirable to have means for editing parameters. So, a simulation simulator should, as do all simulation systems, have functionality to manage parameters.

Finally, the simulator should allow the integration of external tools for complex tasks. For example, the performance prediction may depend largely on accurate predictions of network performance. Network simulation is a major application area of simulation (e.g. SSF, see section 2.3.3), so a network simulator could be used to obtain accurate predictions for message transfer times. All in all, a general purpose simulation simulator requires:

- Powerful interfaces for extending and enhancing the tool via a plug-in mechanism
- An interactive mode to facilitate debugging and observation of single simulation runs
- Visualisation methods to facilitate observation and debugging
- A batch mode to evaluate regions of the simulation space instead of single elements
- Flexible data observation mechanisms
- Parameter management
- Interfaces to integrate external tools

4.3 Shortcomings of the PDES-MAS Simulator

Since SimSim was developed using some code and functionality of the PDES-MAS simulation tool, it is useful to describe advantages and disadvantages of this software. In combination with the identified requirements, these shortcomings determine the major development goals for SimSim.

The PDES-MAS simulator was used by Ewald et al. to evaluate routing algorithms for different types of data access queries in PDES-MAS (see section 2.3.2) [30]. It is tailored to the needs of this goal, so first of all it is *not* a general-purpose PDES simulator.

To simulate the PDES-MAS system, it is modelled as a data structure that contains information about the structure of the CLP tree and existing SSVs. Each SSV can be identified by a unique identifier, the SSV ID, and has a type and a value field, just like real-world SSVs in PDES-MAS have. In contrast to them, the modelled SSVs do not have to contain state-saving information (i.e. information about their former values and access history), because roll-backs are *not* modelled. In addition to the fields similar to original SSVs, each SSV model stores its current position in the CLP tree model.

Since PDES-MAS is used to simulate multi-agent systems, the application model is formed by a set of agents models. Each agent model generates range or ID queries for different SSV types and values. Now, the PDES-MAS simulator analyses each of the queries and *calculates* the number of messages that need to traverse through the tree in order to resolve it. For each query type, this was done by a single function. While this is a very efficient way of forecasting the number of messages that occur in the whole system, the abstractness of this approach brings some problems.

Some additions to the model, for example a data management mechanism (as proposed in [73]), can be integrated easily with it. Such a mechanism migrates SSVs between the CLPs to optimise the resolution of queries. This is done from time to time, and it results in new SSV positions. It can be integrated with the PDES-MAS system model by simply calling a subfunction that adjusts the position variables of the modelled SSVs.

Unfortunately, this is not that easy for other mechanisms, or for other questions to be asked. Consider the question about *network load* instead of message number. Of course, the number (and size) of messages indicates how much *overall* network load is induced, but maybe this network load is unevenly distributed, so that the whole system suffers from slowdowns due to network bottlenecks, *although* having a low overall network load?

Another example is the integration of different synchronisation mechanisms. Even though roll-backs can be detected in principle, it is crucial to identify all ALPs to which a roll-back has to be propagated. Otherwise, the roll-back cost could not be determined and thus the impact on the performance could not be measured. However, this problem is rather hard so solve, since the order in which the queries arrive at each CLP is unknown.

Both difficulties are caused by two shortcomings of the PDES-MAS simulator:

- Weak concept of LPs: The message calculation does not logically separate the LPs. Actually, CLPs are not modelled at all, the simulator just calculates paths in a tree, given a set of SSVs, a routing algorithm, and a query. There are neither explicit data structures for CLPs, nor is there any functionality associated to them. This allows a very efficient execution, but it also hampers the use of the simulator when the behaviour of the CLPs is getting more complex.

- Weak concept of time: Unlike a usual simulation system, the PDES-MAS simulator does not employ a notion of time thoroughly. Although the application model fetches all generated ALP queries per application model time step and then forwards them to the query analysis, there is no notion of simulated wallclock time at all. Consequently, it cannot be measured *when* a query is returned to the ALP, a message occupies the network interface of an LP, or a roll-back occurs.

Therefore, the PDES-MAS simulator is a rather abstract way of simulating a PDES system, it could even be regarded as an analytical approach that just uses some simulation techniques for easier calculation. This abstractness caused some problems during its development. It is very hard to ensure the correct functioning of algorithms that just *calculate* the necessary number of messages, since errors in the calculation only result in wrong figures, not in runtime errors. This means, that one cannot rule out false positives (i.e. the predicted algorithm performance is good, but the simulator does not count all messages) or false negatives (i.e. the predicted algorithm performance is bad, because the algorithm counts too many messages) in principle.

This made a thorough and longish testing indispensable, which led to the development of a post-mortem simulation run visualizer for PDES-MAS. It can be used to observe the behaviour of calculation algorithms, which are the system model in this case, and to control their results manually. Still, debugging and validation were very time-consuming. Moreover, the algorithms to *calculate* the number of messages differ strongly from the routing algorithms *themselves*. So, code for the system model cannot be ported easily to the original system. This is another drawback.

Nevertheless, the PDES-MAS simulator already fulfils some of the requirements that were stated in section 4.2. There are visualisation methods that facilitate debugging, albeit uncoupled from the actual simulation run, and thus not able to interact with it. Besides that, the PDES-MAS simulator has a batch mode that allows the execution of many simulation runs. Additionally, the basic parameter handling via

XML files and *Java Beans* turned out to be quite useful (see section 4.6), although the definition of the actual parameters had to be done by editing the files manually and was rather tedious.

Despite these useful approaches to fulfil some of the aforementioned requirements, it has to be highlighted that to overcome the described shortcomings, the structure of the software had to be completely redesigned. The useful pieces of the PDES-MAS simulator were strongly adapted and redesigned as well, so that they now fit into SimSim's plug-in scheme, which will be detailed in the next sections.

The abstractness of the PDES-MAS system model and the lean and specific design of its simulator also ensured that the execution of the model was quite efficient. It has yet to be shown how SimSim's performance relates to this efficiency (see chapter 6).

4.4 A Generic Model of PDES Systems

Now, it is important to specify a *generic system model* that can be employed in SimSim. Basically, this is the part of a system model that is common for *all* PDES systems to be simulated. Thus, it has to be abstract enough to include all kinds of PDES systems, while being concrete enough to allow the development of a simulator to execute it. Besides that, it should be flexible enough to allow a user to leave out the parts of a PDES system that are irrelevant for task to be solved. In contrast, the term *specific system model* refers to the parts of a system model that model the *specific* properties of a PDES system.

As described in chapter 1.2.3, Misra's notion of logical processes (LPs) that exchange events is fundamental in the context of distributed simulation. Hence, it can serve as a good starting point for further considerations. Evidently, an LP needs to be executed by a certain physical processor (PP). The physical processor, in turn, is connected to other physical processors via a network of any kind. Each LP is managed by exactly one *host processor*, whereas each PP may host an arbitrary number of LPs.

The execution speed of an LP is determined by itself, i.e. the complexity of its algorithms, and the computing power of its host processor p_i. This could be modelled by letting the LP specify its execution time regarding some *baseline system*, and then multiplying this time by a processor-specific *computation power factor* cp_i that determines its computing power. Although this model of execution time is not accurate when it comes to PPs that are equipped with special hardware, so that a PP's specific computation power factor may vary with respect to the *kind* of operations the LP executed, it should suffice in this context. If not, it is also possible to define different factors for different operations, and let each LP define how much time it spent executing one or another kind of operation.

While an LP itself represents a part of a PDES system, its behaviour is determined by the part of the application model it simulates. When another LP has to be notified, an LP generates a message and passes it to its host processor - just like a usual program, which would call a low-level routine that is provided by the operating system or an additional communication layer.

The current *CPU* and network load of host and destination processor, as well as network characteristics and protocols, determine at which simulated wallclock time the event reaches the processor of the LP for which it was intended. This process consists of two parts: At first, the sending processor has to prepare the message and send it. Secondly, the transfer might cost additional time, due to a bottleneck situation at the receiving processor or protocol specifics.

These factors are very complex and difficult to model, so that a modified version of the communication cost model of Juhasz et al. (see section 3.2.2) is used instead. Since some applications of SimSim might need a more sophisticated network model, a *hardware model* is introduced that can serve as an interface to an appropriate network simulator.

As in reality, the physical processors form a logical network, which is a (connected) graph, i.e. each processor manages a list of neighbour processors, to which it has a direct network connection. Thus, it is *not* assumed that each processor has general knowledge of all other processors in the system. This allows to model PDES systems that are executed on a *dynamic* network of processors, e.g. by using *P2P* or grid-related mechanisms.

In SimSim, a processor may *only* send a message to one of its neighbours. More complicated routing strategies that send messages over different intermediate nodes have to be implemented on top of this basic scheme, if need be. This is relatively easy, because one can either enhance the LP logic with an explicit routing algorithm (as it is done for PDES-MAS), use a graph algorithm to approximate the network cost for a custom hardware model, or plug in an existing network simulator. Other, more abstract simulations might also assume a complete graph between the processors to totally circumvent this issue.

This restriction to messages between direct neighbours reduces the communication cost approximation of Juhasz et al. (equation 3.6 on page 38) to $T_{comm} = T_s + k \cdot T_w$, where T_s is the time to initialise a message of k bytes, and T_w is the additional time per byte. In SimSim, the parameters T_s and T_w can be set for *each* connection between two processors (as opposed to the approach of Juhasz et al., who use them as general parameters). Therefore, it is possible to simulate a PDES system's performance under very different circumstances, from efficiently connected clusters of processors to only sparsely connected networks with a high latency.

This allows, in combination with defining an individual computing power factor for each processor, the simulation of PDES system execution in a grid environment. The model of an information interchange between two parts of the application model is illustrated in figure 4.3.

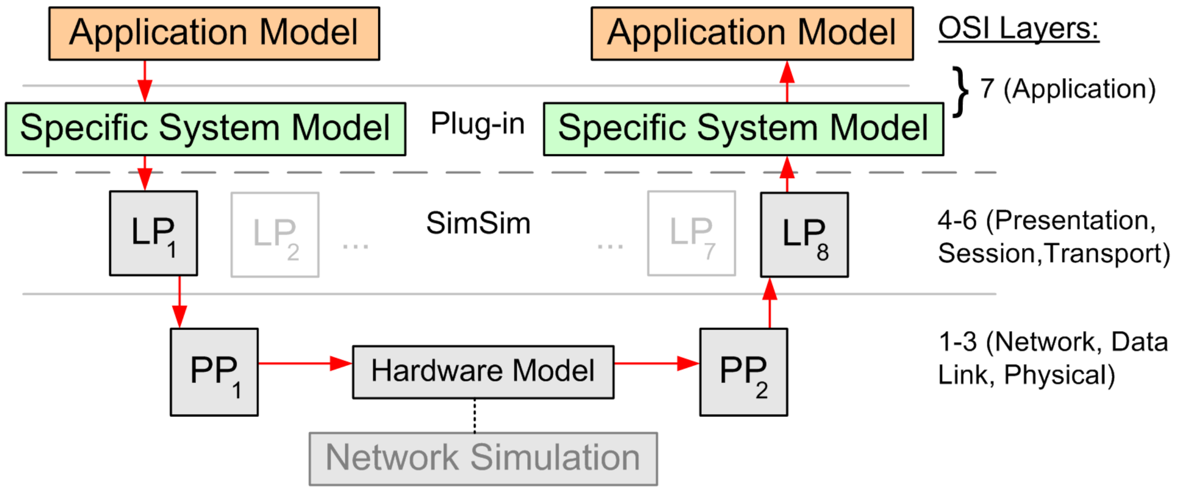


Figure 4.3: Communication Between Application Model Entities. The colours correspond to different levels of the model again (see figure 4.1), except for components with a grey background, which represent the *generic* components of the system model.

As can be seen in figure 4.3, communication in the generic PDES system model strongly resembles to the layers of the OSI reference model [109]. The impact of the media layers (1-3) is modelled by the set of PPs and the hardware model, which manages the PPs, models the interconnecting network, and may serve as an interface to a network simulator. The upper layers are represented by the LPs (except for the application layer).

Actually, LP models do not have to provide any functionality of layers 4 to 6, since this is already ensured by the fact that the communication is simply done by Java method invocations. Nevertheless, in *real* PDES systems, the LPs and the libraries they are using (like MPI or RMI) are very well providing these functionalities.

Figure 4.3 also illustrates which parts of a running PDES system simulation are part of the generic PDES model. Clearly, the notion of LPs is the 'least common denominator' for all kinds of distributed simulation systems. In fact, this model is general enough to include *any* distributed system, since such a system simply consists of a number of LPs that are hosted on processors and exchange messages.

Each LP has to be attached to its part of the system model, which in turn is associated with a certain part of the application model. Both system and application model have to be provided by the plug-in, because both can differ strongly for different applications of SimSim. Thus, the LP models are the main interface of SimSim to its plug-ins.

Besides motivating the general structure of the generic PDES system model, it is also important to consider how the actual simulation of it should look like. From an exterior view, the system model merely consists of some LPs that are sending messages to each other. Each message is sent and received in simulated wallclock time. When an LP receives a message, it may react to it by letting its host processor schedule a number of new messages to SimSim's central event queue. When the LP has finished, SimSim removes the event with the smallest time stamp from its event queue and passes it to its destination processor.

The problem is now to determine the arrival times of newly scheduled messages. This cannot be done by just considering the computing power factor of the host processor, or the time that is needed to send a message over the network.

Additionally, one has to account for the possibility of *multiple* logical processes that are hosted on one physical processor, as well as the computational load each of these processes might induce. All LPs that are hosted on one PP are competing for a single resource, which is the CPU of the physical processor. Hence, the arrival time of a message depends on the status of the sending physical processor. If it is overloaded and the sending LP has to compete with many other LPs for computing time, the creation (and thus the arrival) of all its newly scheduled messages is likely to be delayed.

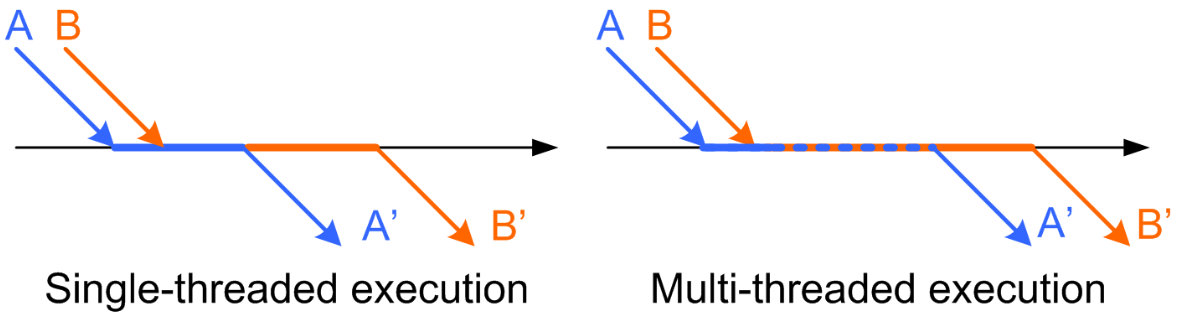


Figure 4.4: Single- and Multi-threaded Execution: Depending on the way different LPs (blue and orange) are executed on one processor, they need different time spans to react to a message (A, B) by sending a new one (A', B').

Furthermore, there are different strategies to handle multiple threads. A variety of solutions has been developed for operating systems [83], which usually have to handle many processes at the same time. Figure 4.4 illustrates two opposed strategies of CPU allocation, and how they influence the point in time at which a message is eventually sent. On one hand, messages could be processed one after another, and a thread that currently processes a message is not interrupted when a new message arrives.

On the other hand, the processor could execute *multiple* threads simultaneously, each of them processing the message it received. While multi-threading support might be desirable for operating systems in general, there are several reasons why this way of execution is very hard to model: For one, the management of different threads may induce a considerable overhead [42], which depends on the actual scheduling mechanism. This has to be modelled as well. Besides that, the moment in which a thread has finished and finally sends a message is very hard to calculate: It might be easy to calculate the overall processing time of a thread on a multi-threaded system for a *static* situation, but during this processing time, *new* messages might arrive at the processor, which may activate additional threads and thus influence the processing time. This problem exists for each of the parallel running threads, so that it is hard to determine for which duration a thread shares the CPU with which amount of other threads. So, calculating a thread's processing time on a multi-threaded system is not trivial.

Therefore, SimSim's generic PDES system model assumes a single-threaded execution on each processor. This assumption is not restrictive, since the general effect of a multi-threaded execution might still be modelled (e.g. by setting the computing power factor cp_i accordingly), albeit not calculated accurately. It is also still possible to model multi-processor systems, by adding a node for each processor to the hardware model and inserting edges for *very* fast communication between them. Moreover, for many PDES system implementations (e.g. James II), it is actually preferred to let a single thread handle all messages, since threading, as already mentioned, can be quite costly.

In contrast to multi-threaded execution, the thread execution time for single-threaded execution is relatively straightforward to calculate, so that it suffices to consider the event with the smallest time stamp (see section 4.6). Additionally, it is assumed that an LP sends all messages *after* processing the current message. This assumption can be circumvented by letting an LP send messages to itself, which 'splits' the modelled processing time into several parts. After each part, new messages can be sent.

Now that a suitable system model has been worked out, as well as an approach to conveniently simulate it, one may ask how the rest of the system, i.e. the specific PDES behaviour and the application model, can be modelled. A very popular formalism to model software is the Unified Modelling Language (UML) [13], and particularly its State Charts diagram type, which allows the definition of program behaviour. Of course, many other discrete-event modelling formalisms, e.g. DEVS, could be used to specify system and application model.

Since identifying or developing a modelling formalism that suits well for the specification of PDES algorithms is quite complex, SimSim does neither restrict nor support the use of any modelling formalism. Instead, it lets the programmer of the plug-in choose freely the way to define the needed behaviour. In fact, it might even be the case that simply *programming* each new algorithm turns out to be a quite feasible approach. An implementation that works on the rather abstract level of the generic PDES system model is in fact a *model* of the algorithm to be evaluated. Most likely, the development of such a model will be faster and easier than the development of the actual algorithm. It allows the developer to abstract peripheral mechanisms away, and to focus on the most important aspects (as exemplified in section 5.4). Moreover, the resulting code could possibly be re-used for an implementation of the real algorithm (see section 5.6). Besides that, abstract modelling of the software might be quite laborious, and the learning curve for a new modelling formalism might by correspondingly steep. Nevertheless, this issue deserves some further considerations (see section 7.3).

4.5 Design of SimSim

This section describes and motivates the major design principles that were used to develop SimSim. SimSim's design has to meet the requirements listed in section 4.2, and has to take the structure of the generic PDES system model into account as well (see section 4.4). Finally, it was important to overcome the problems that hampered the flexible application of the PDES-MAS simulator (see section 4.3). Appendix B provides supplementary UML diagrams.

4.5.1 Overall Composition and Core Classes

The overall composition of SimSim is outlined in figure 4.5. It shows how the three major parts of SimSim – the user interface, the simulator and the plug-ins – are related. The differentiation of user interface and simulator follows two purposes: For one, this design allows the implementation of *different* user interfaces for the simulator. Basically, any class that implements the interface `IUserInterface` may serve as a user interface.

Another purpose is to let SimSim fulfil the requirements concerning visualisation and an interactive simulation mode. To support interactive simulation, it is important to let the user stop or pause the simulation at any time. This can be realised by implementing user interface and simulator in different threads, which in turn is facilitated by entirely decoupling user interface and simulator.

As also outlined in figure 4.5, the plug-ins form the third major part of the software. They define specific system and application models. Since these might need parameters, plug-ins have to provide custom

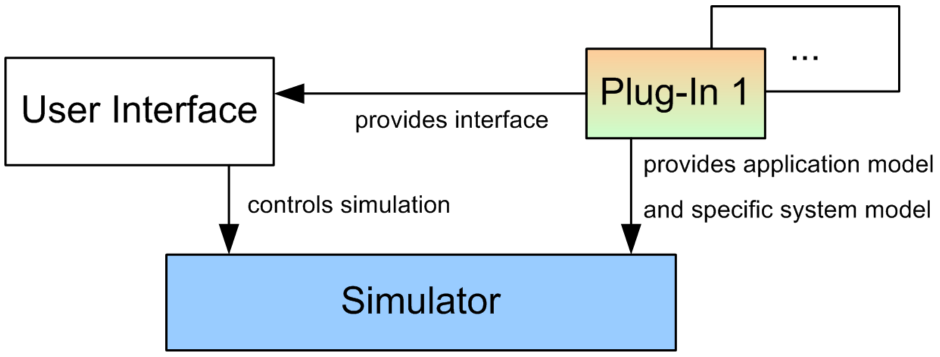


Figure 4.5: SimSim's Basic Parts: SimSim is mainly divided into a user interface and a simulator component. Plug-in components have to pass the specific system model to the simulator and provide supplementary interfaces to the general user interface. Again, the colours denote the relationships to the parts of a PDES system model (see figure 4.1). The colour gradient highlights that that both the specific system model and the application model have to be provided by the plug-in.

configuration dialogues.

The design of the core simulator classes is straightforward, since the simulation procedure, as already described in section 4.4, simply consists of a loop that lets the destination LP process the event with the minimal time stamp. The core classes are depicted in figure 4.6.

The class performing the simulation is `SimSim`, which holds an `EventQueue` object and executes the events one after another. It does so by calling the host processor of the LP the current event is intended for. Events are instances of the class `MessageEvent` (not depicted in figure 4.6). If the event queue is empty, the simulation stops.

An event has a time stamp that defines the point in simulated wallclock time at which it arrives at the destination processor. Additionally, the event stores references to source and destination LP, as well as an instance of `AbstractMessage`. This is an abstract class that has to be subclassed by plug-ins, to represent the messages in the system model they provide. So, `AbstractMessage` exploits *polymorphism* to serve as a placeholder for any of its subclasses.

`SimSim` retrieves the event with the smallest time stamp from its `EventQueue` object, and passes it to the host processor of its destination LP, which then propagates its message object to the destination LP. To call the physical processor instead of directly passing the event's message to the destination LP is necessary for the creation time calculation of newly generated messages, as described in section 4.6.

Each Logical Process is associated with exactly one instance of `DistributedSimulationProcess`, which again is an abstract class that has to be subclassed by a plug-in (see figure 4.6). When a logical processor is called to receive a message, it propagates the message to its instance of `DistributedSimulationProcess`. Such instances will be named *system model processes* in the following.

The underlying *software design pattern* is also known as the strategy pattern [38, p. 315 - 323]. It is used to *"Define a family of algorithms, encapsulate each one, and make them interchangeable."*[38, p. 315]. Here, the algorithm to be encapsulated defines the behaviour of the PDES system, which is unknown at design time and thus has to be separated from the rest of the software. In general, the LP's call to the `DistributedSimulationProcess` object can be regarded as the switch from generic to specific system model. A system model process is initialized with a reference to the LP it is associated with. `LogicalProcess` objects provide a public method to send new messages, and propagate them to their host processor object. The physical processor is now able to determine the time at which the message will reach its destination, and thus adds a new event to the event queue.

In principle, one could also directly subclass `LogicalProcess` to implement a specific system model,

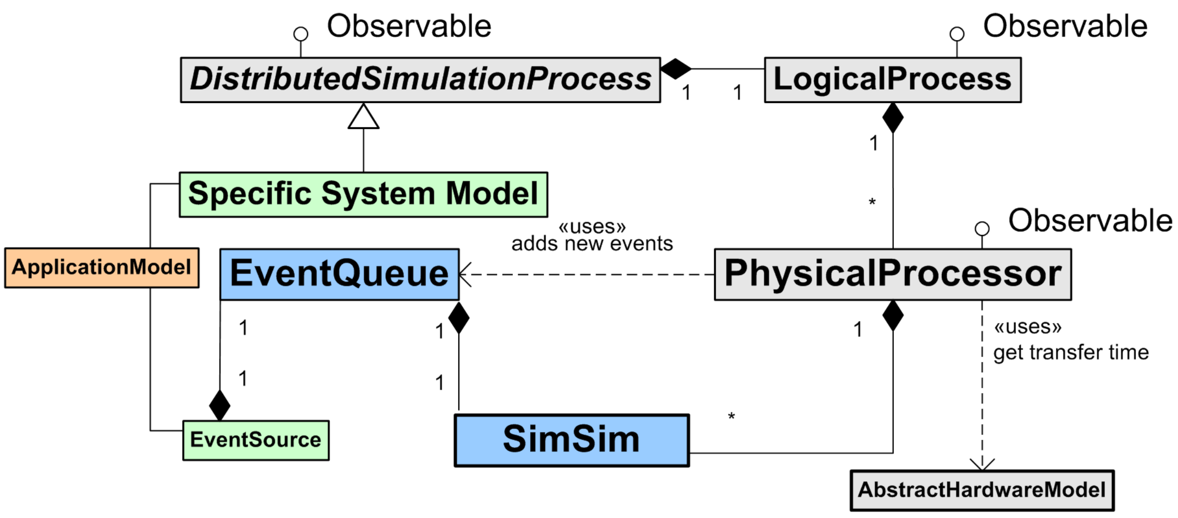


Figure 4.6: SimSim's Core Classes: The core class, `SimSim`, is equipped with an event queue and calls the corresponding physical processor to process the next event. Application model and specific system model are obviously *not* part of the core classes, but illustrate where the plug-ins come into play. Some classes implement the `Observable` interface, which will be explained in section 4.5.2. The colouring represents the different parts of the PDES system simulation (see figures 4.1, 4.3).

but the strategy pattern realises a looser coupling between specific and generic system model. This has the advantage of being also extensible on the client side, i.e. the generic system model can be altered without affecting the specific one. So, it is possible to implement more detailed models of LPs, possibly modelling memory consumption or certain particularities regarding the operating system, without having to change anything else.

The class `AbstractHardwareModel` serves as an interface for external network simulators and is used by the physical processors to determine inter-processor transfer times. Finally, on the left side of figure 4.6, the basic structure of a plug-in implementation is outlined. A plug-in has to provide at least one subclass of `DistributedSimulationProcess`. There may be different types of LPs, like ALPs and CLPs in PDES-MAS, which demand further differentiation. Additionally, a plug-in needs to provide an object that serves as an *event source* and implements the corresponding interface.

Primarily, an event source is needed to start the simulation run, since the event queue is empty at first. Hence, the event source is the source of the first event. In some situations, e.g. for PDES-MAS, the event source can also be used to generate new events continuously (this mechanism is detailed in section 4.6). In the PDES-MAS system model, only the CLPs are instances of `DistributedSimulationProcess` and associated with LPs, the ALPs form the application model and are issuing their queries via the event source mechanism. In contrast, the event source for James II *only* triggers the simulation run, it does not add any new events. Additional messages would invalidate the execution of the abstract PDEVS simulator.

In general, it should be noticed that the arrangement of the core classes was strongly influenced by the structure of the generic system model. Hence, all parts of the generic system model (hardware model, logical process, physical processor) have been manifested in distinct classes. Moreover, some supplementary components have been added, like the event queue or the abstract classes. The use of the strategy pattern facilitated the separation of concrete functionality and concrete data from their abstractions. Besides that, the way in which the generic system model handles an event, by passing it from PP to LP, and from LP to a custom plug-in object, is in itself a software pattern named 'chain of responsibility' [38, p. 223 - 233]. It results in *"added flexibility in assigning responsibilities to objects"*[38, p. 226], which is needed to support the plug-in interface, one of SimSim's main design requirements.

4.5.2 User Interface

The user interface of SimSim should be easy to use and easy to extend at the same time. To improve the usability with respect to the PDES-MAS simulator, a small subsystem to edit parameters via dialogues has been implemented.

It consists of a small class hierarchy of `AbstractParameterPanel` subclasses, which allow the specification of constraints, and an `AbstractParameterDialog` class to which any number of these panels can be added (see figure 4.7). A dialogue that edits an object's properties can be created by inheriting from `AbstractParameterDialog` and adding appropriate parameter panel objects.

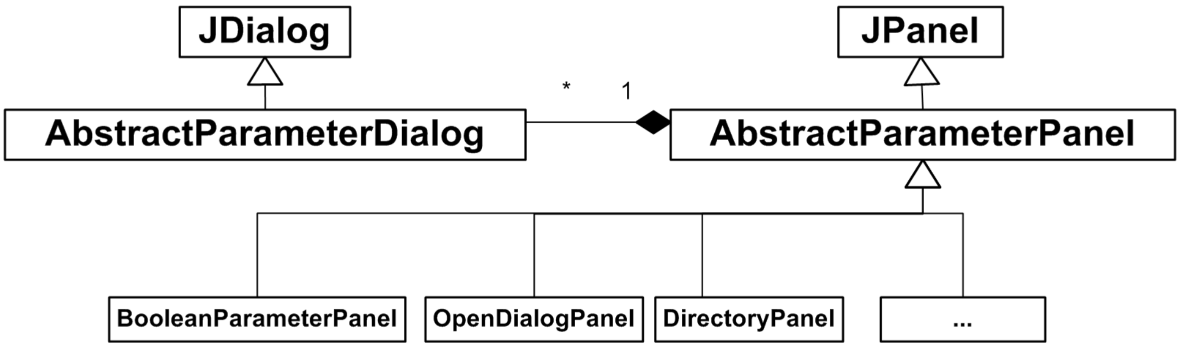


Figure 4.7: SimSim's Classes to Support Graphical Parameter Editing

Again, a polymorphism is used to ensure the system's extensibility. Besides means to edit primitive types, such as `boolean` or `double`, there are also standard dialogues that allow the definition of probability distributions, panels to manage lists of objects, or others that let the user select a folder in the file system (e.g. the target folder to store results in). Constraints can be defined by adding objects that implement the `IConstraintWatcher` interface to the panel objects. There are some standard constraint watchers that can be used by a plug-in programmer, e.g. for ensuring that a value is defined within a certain interval (inclusive or exclusive). When a constraint is violated, an appropriate default value, for instance the lower bound of an interval, is set. Furthermore, the user gets a graphical notification of the constraint violation.

The code of `AbstractParameterDialog` and `AbstractParameterPanel` also handles the refreshing of the user interface components, since the values of some variables might be changed externally (e.g. by SimSim or the plug-in). This simple framework is used to define all dialogues in the system. It drastically lessens one's efforts to program suitable dialogues.

In addition to editing and managing parameters, the user interface is required to display visualisations of running simulations. This implicates that the visualisations have to be *synchronised* with the current simulation run. If not, the user might try to pause the simulation in order to scrutinise a certain situation, but the simulator has already advanced to another state, so that this feature becomes useless.

One could imagine a broad variety of very different visualisations for one and the same simulation run, e.g. by focusing on the physical processors, the logical processes, or the application model. While some visualisations, like network traffic between PPs or LPs, are often useful and sufficiently general to be provided by the main user interface, other visualisations are not. For example, all visualisations regarding the application model are plug-in specific and have to be implemented by the plug-in.

Therefore, the number of visualisations is unknown and may vary from plug-in to plug-in. To synchronise this unknown number of visualisations, a barrier synchronisation is used. All visualisation components have to register at the user interface. Only when each of these components has eventually animated the event that was processed, the barrier synchronisation is released and the simulator can continue.

Evidently, a barrier synchronisation might induce a considerable overhead, as well as various visualisations that may extensively animate each event. While this is still useful for debugging or the analysis of specific situations, it might be desirable to avoid this overload when a multiplicity of simulation runs should be executed. So, the user interface has to differentiate between interactive and batch execution. For the interactive execution of a simulation run, the above mentioned features are used to ensure its correct functioning. The batch mode, instead, is simply visualised by two progress bars that indicate which parameter combinations have already been simulated.

Finally, a way of observing the central simulation components is needed, so that the visualisation components are actually able to display certain aspects of them accurately and immediately. This is a very common problem for software systems in general, and particularly in the domain of modelling and simulation. There is a classic solution for this problem, defined by the so-called Observer pattern [38, p. 293 - 299]. The realisation of this pattern in SimSim is quite simple. All components that should be observable are implementing the `IObservable` interface. This interface provides functions for all objects implementing the `IObserver` interface, so that they can register themselves at the observable objects. If something extraordinary occurs at an observable object, e.g. an LP receives a message, it notifies its observers of this event. This mechanism is used to store result data and to provide the visualisation methods with the information they need. As can be seen in figure 4.6, most of the core classes are equipped with an `IObservable` interface. Some screenshots of the user interface can be found in appendix C.

4.5.3 Plug-Ins

SimSim's interface to support plug-ins is crucial to its applicability. If the interface is too complicated and too hard to use, the learning curve for using the simulator might be too steep to gain any benefits from modelling a PDES system in advance.

On the other hand, SimSim should bring as much features to support the plug-in developer as possible. Again, the strategy pattern is used to allow a convenient extension of the software: A plug-in has to subclass the abstract class `SimSimPlugIn`, which declares methods for initialising, resetting and configuring the plug-in. Each plug-in has exactly one subclass of `SimSimPlugIn`, which is also called the *main plug-in class*, since it is the central interface of the plug-in, behind which the rest of its code is 'hidden' from SimSim.

Consequently, the strategy pattern is used to decouple any point of contact between plug-in code and SimSim code. Abstract classes and interfaces define the methods that SimSim calls. Concrete objects for them are retrieved by calling methods that are defined in `SimSimPlugIn`. One could also regard this structure as a factory method pattern [38, p. 107 – 116], because the main plug-in class serves as an object creator (i.e. a factory) for all objects that depend on application model or specific system model properties, and thus depend on the concrete plug-in implementation.

For example, there are many parts of the user interface that rely on plug-in implementations. SimSim requires a main visualisation panel for interactive simulation, which has to be a subclass of `AbstractVisualisationPanel`. As another example, custom rendering methods to appropriately display the state of SimSim's event queue are needed, since the events can be distinguished by the type of message they are associated with. The necessary functions are declared in the interface `IEventTableRenderer`, which has to be implemented by a plug-in class.

This mechanism was used quite often, but many functions of the main plug-in class are *either* used for interactive simulation *or* to support a batch execution. So, it is desirable to divide these operations and to define two interfaces that mark a plug-in as capable of interactive simulation and batch mode simulation, respectively. This makes it possible to support only *one* of these modes at first, while already having a running system to test.

The interfaces are called `IBatchModeSimulation` and `ISimulation`. `IBatchModeSimulation` declares methods to edit and interpolate an arbitrary Java object, so that the kind of interpolation (e.g. linear, etc.) can be freely defined by the plug-in developer. Moreover, the plug-in can also define interpolation routines for more complex objects, like vectors or matrices. Basic interpolation routines are provided by an auxiliary class, so that the developer only has to take care of more specific demands.

In contrast, `ISimulation` declares different functions that return objects for visualisation. Again, the developer only has to implement the specifics of its plug-in. If only the *results* of a single simulation run are interesting and therefore no visualisation is needed, one could subclass `AbstractVisualisationPanel`

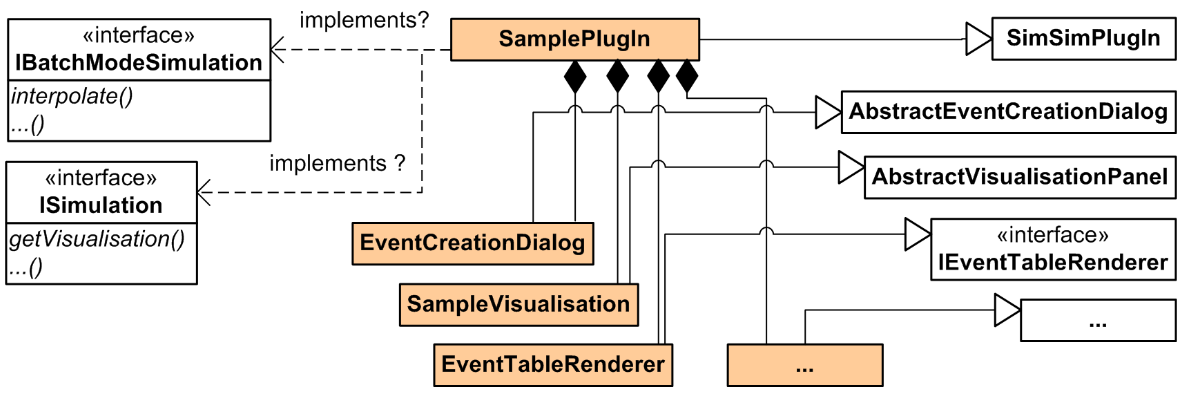


Figure 4.8: SimSim's Plug-In Interface: The classes with orange background are the ones that need to be implemented by a plug-in developer.

without adding new code, which will result in a blank visualisation panel. Nevertheless, the system will work by then. `AbstractVisualisationPanel` also supports the plug-in developer by providing some higher-level visualisation methods (see section 4.6). A sample plug-in class structure, in conjunction with SimSim's plug-in interface, is shown in figure 4.8.

4.6 Implementation Details

SimSim is written in the Java programming language version 1.5 [48], using the open-source development environment Eclipse [47]. This section illustrates some more complex development problems and their solutions.

Plug-In Initialisation

Before a plug-in can be used, it has to be initialised properly. The main task during initialisation is to instantiate custom subclasses of `DistributedSimulationProcess`, and to associate them with logical processes. To do so, an instance of the main plug-in class is called with a reference to the main `SimSim` object. It may then call the `SimSim` object's `createLogicalProcess()` method to obtain the LPs to which it can register its custom simulation process classes.

The spawning of a new LP and its assignment to a physical processor is done by a subclass of `AbstractHardwareModel`. A plug-in may set a custom hardware model during initialisation, otherwise the `DefaultHardwareModel` is used, which assumes that each LP runs on a single processor and thus 'creates' a new physical processor each time a new LP is created. `DefaultHardwareModel` further assumes a fully connected network topology (i.e. a complete graph), and implements the modified communication time model of Juhasz et al., as described in section 4.4.

To support batch mode execution or successive interactive simulation, each plug-in also implements a `reset` method, which re-initialises the plug-in after a simulation run.

Event Queue Implementation

The event queue is the central data structure of SimSim, and therefore needs to be described in a more detailed way. Its performance is crucial to the overall system performance, because queuing and dequeuing events are the simulator's most complex operations during the simulation run. All other operations perform either basic arithmetic calculations, method calls to plug-in objects, or file access, so that they do not have a great potential for being optimised.

However, since the efficient implementation of event queues is quite complex and well beyond the scope of this thesis, a simple linked list, which uses binary search to insert new events, was used.

The event queue execution may also be hampered by its obligation to notify all registered objects about state changes. This has to be done when running a simulation interactively, so that the user has knowledge about all scheduled events. Additional overhead is avoided by putting this functionality into a subclass of `EventQueue`, `InteractiveEventQueue`, which is used instead of `EventQueue` when SimSim runs in interactive mode.

Calculation of Event Times

The calculation of the event time, as already mentioned in section 4.4, is not trivial. Even if the described assumptions hold true, which means that all LPs send their messages *after* they completely processed a received message, and that all physical processors work single-threaded, a physical processor still needs to consider different measurements to solve this task.

After an event is received and the contained message is passed to the corresponding system model process, this has to define how much time it *would* have needed to process this message. As described in section 4.4, the returned value has to be normalised, so that it can be set in proportion to the actual processing power of the host processor. Hence, the value defines the processing time a system model process *would* have needed in reality, running on a *specific* baseline system. It will be multiplied by the computing power factor cp_i. For example, cp_i is set to `0.5` if processor i is twice as fast as the baseline system.

Algorithm 1 Event Time Calculation of a Physical Processor

```
class PhysicalProcessor {

boolean noMsgSentAfterRecv ← true
double busyTime ← 0.0
double cp ← 1.0

function  receiveMessage(MessageEvent e)
  busyTime ← max(busyTime, e.getTime())
  noMsgSentAfterRecv ← true
  e.getDestination().receive(e.message, e.source)
  if(noMsgSentAfterRecv)
    busyTime ← busyTime + e.getDest().getPTimeRecv() · cp
return

function  sendMessage(Message m, LP source, LP dest)
  double sendTime ← source.getPTimeSend() · cp
  if(noMsgSentAfterRecv) {
    sendTime ← sendTime + source.getPTimeRecv() · cp
    noMsgSentAfterRecv ← false
  }
  busyTime ← busyTime + sendTime
  double addT ← 0
  if (dest.getHost() ≠ this)
    addT ← hwModel.getNWTime(id, dest.getHost(), m.size())
  addEvent(new Event(source, busyTime + addT, dest))
return

 ...
}
```

Besides the time that is needed to process a received event, a system model process might need some additional time to create a *new* message, because it may be quite large or complicated to generate. This has not to be confused with the time for the *network interface* to set up a message, instead it adds up to the processing time of the process.

Both values, the modelled processing time for receiving the last event, named *receive time*, and the time that was needed to sent the next event, named *send time*, have to be calculated *before* any new message is sent. Otherwise, the host processor is not able to calculate a time and schedule the new event. A pseudo-code algorithm that shows how a physical processor is able to determine the event scheduling time is presented in algorithm 1.

A physical processor uses two methods to calculate the event time of new messages: `receiveMessage(...)` will be called by the simulator to process the next event, and `sendMessage(...)` is used by a system model process[1] to send new messages.

Each processor maintains a *busy time*. This is the point in simulated wallclock time until which the processor's CPU will be busy. If a processor receives a message and was idle before (i.e. its busy time is smaller than the current simulated wallclock time), the busy time is set to the received event's time stamp, which in fact *is* the current simulated wallclock time. Then, a flag named `noMsgSentAfterRecv` is set to `true`, and the message contained in the event is passed to the event's destination LP.

Now, the received message is processed by the system model process, which may send new events to other LPs. If it does, the `sendMessage(...)` method is called. Here, the processor firstly requests the send time for the new message, which has to be scaled by its computing power factor `cp`. If no other messages have been sent yet, which is determined by the aforementioned flag `noMsgSentAfterRecv`, it is necessary to add the receive time as well. To ensure that this is just done once, the flag will then be set to `false`. Afterwards, the `sendTime` is added to the busy time of the processor. If the message is not intended for an LP on the same host, the processor additionally lets the hardware model calculate its network transfer time. Finally, the message is scheduled with a time stamp that is formed by adding up the current busy time of the processor (i.e. the time at which it was sent away) and the network transfer time (i.e. the time it took to reach the destination).

If another message has to be created and `sendMessage(...)` is called again, `noMsgSentAfterRecv` has already been set to `false`, so that the receive time is not added to `sendTime` again. Eventually, the system model process will finish, and the execution jumps back to `receiveMessage(...)`. Now, the processor has to test whether the system model process has sent any message. If not, i.e. `noMsgSentAfterRecv` is `true`, it still has to add the receive time to its busy time. Figure 4.9 illustrates the situations with which the algorithm has to cope.

The algorithm implicitly assumes that each processor is *always* able to receive messages without interrupting its current thread, and that it is also able to queue messages without memory restrictions. The first assumption might be valid for some computers, or the effect of thread interruption might be negligible. The second assumption is clearly *not* valid, but might be irrelevant in the context of SimSim's application. However, it is still possible to avoid *both* assumptions by enhancing the generic system model. For example, the physical processor could re-schedule events it cannot handle any more, schedule them for the sending processor (i.e. send them back), or increase its busy time in a more sophisticated way. However, both assumptions are deemed to be negligible for the actual task of developing a load balancing algorithm for the abstract PDEVS simulator of James II (which minimises the number of messages, see section 2.3.1), so the event time calculation algorithm was kept as is.

Visualisation

Providing convenient visualisation features is very important, because it lets the plug-in developer focus on modelling the specific system model instead of investing much time for a graphical representation of it. To fulfil this requirement, the log file visualisation component of the PDES-MAS simulator was generalised and enhanced. It employs the Batik SVG Toolkit [98], which allows to display data in the scalable vector graphics (SVG) format.

[1]A system model process calls its host processor's send method indirectly, by calling its own LP.

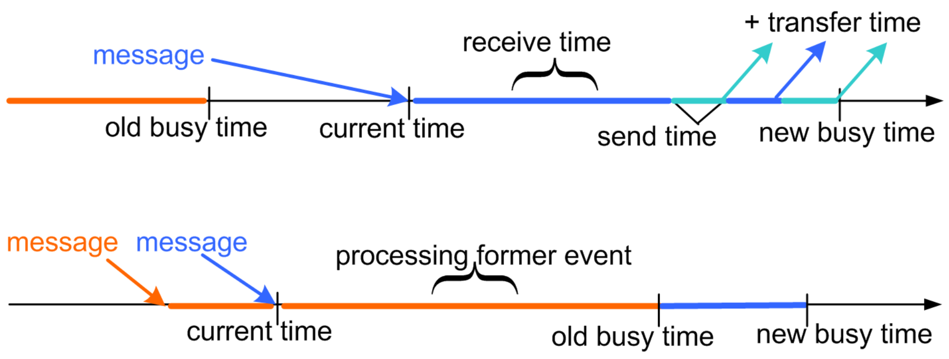


Figure 4.9: Different Situations for Event Calculation: In the situation on the top, the processor is idle until it receives a message. Then, the message is processed (receive time), and three new messages are sent, each of them taking the same time (send time). On the bottom, the processor receiving an event is not idle, but will be busy processing a formerly evaluated message (orange) when it receives a new one (blue). This will be processed after processing the former message has stopped. Old and new busy time denote the busy time values before and after evaluating the blue message.

With SVG, it is very easy to declare and update geometric primitives like rectangles, circles, or lines. Batik provides the `JSVGCanvas` component, which supports continuous zooming, rotating, and panning. Additionally, it is possible to store 'snapshots' of a visualisation to SVG files, so that they can be analysed later. The Batik features have been encapsulated in the class `AbstractVisualisationPanel`, which is a subclass of `JSVGCanvas` and also implements the basic initialisation code to setup an empty document and display it. It can be subclassed by a plug-in developer, which just has to add the code to create the visualisation elements. To support this, `AbstractVisualisationPanel` provides further auxiliary functions. Finally, it supports easy animation by letting subclasses override the methods `animateEvent(...)` and `deanimateEvent(...)` to perform and undo visualisation updates.

Observation and Batch Mode

To supplement the possibility of implementing custom observers for each plug-in, SimSim provides a standard observer that is able to observe events on the level of physical processors. It can be used for interactive and batch mode simulation. In interactive mode, the simulated wallclock times at which messages are received and sent can be stored to distinct files for each processor. As the simulation continues, and thousands of events may be processed, theses files may become quite large. On the other hand, they may allow a *very* precise analysis of the network traffic that was induced by the system model.

In contrast, only the overall simulated wallclock times are stored when running in batch mode. Of course, existing data observation mechanisms can be extended by additional custom measurements. The parameter dialogue to configure plug-in specific observation parameters is integrated into then dialogue to edit the general observation parameters.

To facilitate result interpretation, the predicted execution times of the simulation runs are stored in a single file when running in batch mode. Since the user may choose to vary two parameters, this results in a $n \times m$ matrix. As in a usual diagram, the first dimension will be varied horizontally (the x axis) and the second one vertically (the y axis). The result of the simulation run using the lower border of both parameters is bottom-left, which makes it more intuitively to read. It is then possible to analyse the data with the help of Excel, MATLAB, or other software.

Managing Parameters

Parameter management, which is also a requirement for SimSim, was realised by two independent mechanisms. The parameter editing is done by parameter dialogues (see section 4.5.2), whereas the ability to save and load parameter sets is implemented by the `Serializer` class.

The serialization mechanism could be re-used from the PDES-MAS simulator. It takes advantage of existing mechanisms to store Java Beans to XML files (see section 4.3). However, the existence of a plug-in mechanism and therefore plug-in specific parameters which are unknown at design time demanded some additional efforts.

Since a *complete* set of parameters has to be managed, a main parameter class named `GeneralParameters` is defined. It holds references to different subsets of plug-in specific parameters. Now, plug-ins can define custom classes to store their parameters in. To be manageable by the parameter system, these have to be Java Beans as well.

The PDES-MAS Plug-In

The plug-in to support a PDES-MAS simulation was entirely derived from the PDES-MAS simulator. Nevertheless, almost all of the code had to be re-written or refactored. The reason for this is, as was already mentioned, that an algorithm for *counting* the number of messages induced by a *hypothetical* algorithm is completely different from actually implementing the algorithm, so that it works within the generic system model.

This conversion was done for the address-based and the range-based routing algorithms. Furthermore, a rudimentary graphical animation of the CLP tree was ported from the log file visualiser of the PDES-MAS simulator. The plug-in is executable, and both routing schemes are working, so that it is no problem to add the remaining features of the PDES-MAS simulator (e.g. the integration of the load management scheme by Oguara et al., see section 2.3.2). However, since this is not the focus of this work, the development efforts were then concentrated on the development of a plug-in for James II.

The James II Plug-In

To begin the development of a load balancing algorithm, it was necessary to build a model of the abstract PDEVS simulator, as it is implemented in James II. Some elements of the PDEVS specification and the definition of the abstract PDEVS simulator could be abstracted, which greatly facilitated this task.

For example, coupled models were not treated as independent entities, but their behaviour was completely emulated by the coordinator components that were attached to them. Moreover, the connections between PDEVS models are negligible, so that the notion of ports and couplings could be left out. Instead, each atomic model has a unique ID and may specify the id of the model this message is destined for.

This is still a *valid* model of a PDEVS simulation in James II. Firstly, it can be assumed that the developer has properly defined all couplings that will be used, so that there *would be* a corresponding coupling when a DEVS model receives or sends data. Moreover, multiple couplings can be modelled by simply sending multiple messages. The message number minimisation of the abstract PDEVS simulator implementation in James II ensures that the same amount of messages will be transferred, albeit these messages will have a different size, which has to be taken into account when analysing the result data. Since the application model that is used throughout this work (see section 5.3) will *not* make use of multiple couplings anyway, couplings and ports can be regarded as irrelevant in this context.

Instead of using ports and couplings, the James II plug-in defines the `IModelManager` interface for which an application model, i.e. a concrete DEVS model, has to provide an implementation. This object serves as a repository for all atomic models, and provides methods to 'route' a DEVS message. A coordinator can call the current instance of `IModelManager` to decide whereto it has to propagate a received message. The `IModelManager` instance holds the graph representing the abstract simulator tree, so that it can resolve the message routing.

In addition, a basic load balancing mechanism had to be implemented, which can later be used by a concrete algorithm. To solve this problem, each system model process hosts a *set* of DEVS coordinators and simulators. So to say, another layer for the system model is introduced. It is situated between physical processor and logical processes on one side, and the code that actually reacts to the received message on the other side (i.e. the models of DEVS coordinator and simulator). Now, it is possible to 'migrate' coordinators, simulators and model elements to other LPs. This migration is performed by a `LocationManager` object, which re-sets all references. It stores the physical positions of all elements in the abstract simulator tree, so that they can be visualised easily.

These additional mechanisms for routing and load balancing might seem to be not very helpful and quite complex in themselves. However, they could be implemented very fast and are very easy to debug, since everything is centralised and executed sequentially. The code to model the entire behaviour of the abstract DEVS simulator consists of ≈ 650 lines of code, whereas the *real* implementation of the abstract PDEVS simulator is much more complex and consists of about 2.500 lines.[2]

[2]This was counted by Metrics, an Eclipse plug-in that calculates software metrics [79]. The estimation for James II based on revision 2750. The classes situated in `james.core.model.devs` and `james.core.processor.devs.parallel.conservative.abstractsim` were considered (sub-packages excluded).

5 A Load Balancing Algorithm for James II

In this chapter, a load balancing algorithm for the abstract PDEVS simulator will be developed, aided by SimSim and its James II plug-in. This exemplifies the use of SimSim for the development of new PDES algorithms.

Section 5.1 clarifies the term 'load balancing' and introduces some related work. Afterwards, requirements for a load balancing algorithm that suits to the abstract PDEVS simulator are worked out (section 5.2). A benchmark DEVS model is introduced in section 5.3, so that it is possible to assess an algorithm's performance with respect to the requirements postulated before.

Section 5.4 is the core section of this chapter. It describes the actual development of an algorithm that aims at fulfilling the aforementioned requirements, and illustrates how SimSim facilitated the development process. Additionally, larger-scale experiments to test the algorithm's behaviour more deeply are formulated and analysed in chapter 6.

To show that SimSim is more than a simple sandbox to test an algorithm's basic effectiveness, its ability to accurately predict an algorithms behaviour has to be evaluated. To do so, the benchmark model had to be ported to work with James II, as well as the finalised version of the load balancing algorithm. This is detailed in section 5.6. Again, the results are analysed in chapter 6.

5.1 Related Approaches

As already described in section 2.2.3, load balancing mechanisms are used to manage the load of individual processors in a distributed system. By assuring that no processor in overly slowed down by too much load, the collapse of single nodes is avoided and parallel execution is optimised. Consequently, research on load balancing algorithms originated from distributed (operating) systems.

Nevertheless, there are a *lot* of load balancing algorithms specifically built for PDES. They exploit additional knowledge about PDES systems in order to balance the load more accurately than their more general counterparts.

In [108], Zhou differentiates between load *sharing* and load *balancing*. He argues that the term load balancing implies the equalisation of processor load to be the ultimate goal of such an algorithm, which may not be the case. For example, a PDES-specific load balancing algorithm might try, above all, to minimise the execution time of simulations. This does not necessarily correspond to the goal of having equal load on each processor. In contrast, load sharing could be understood as mere *"redistribution of the workload"*[108, p. 1327].

In the following, the term load balancing will be used in the much broader sense of load sharing, as Zhou already suggested [108]. So, a load balancing algorithm for PDES is simply an algorithm that changes the physical positions of model and simulation parts in order to improve the PDES system's performance. Consequently, this definition also includes algorithms that *do* move model entities, but do *not* attempt to equalise the load at all. An example for such an algorithm would be the 'load management' mechanism for PDES-MAS, as proposed by Oguara et al. [73]. Despite this generalisation, the difference between *initial* positioning of model entities (i.e. partitioning) and *re*-positioning of entities during runtime (i.e. load balancing) has still to be stressed.

Since load balancing algorithms are often complex, there are several studies that assess their behaviour with the help of simulations. Zhou compared seven load balancing algorithms for distributed systems [108]. To do so, he used representative traces of process executions on a single system, which then were replicated. The processors were assumed to be homogeneous, each having two process queues: one for processes running in the foreground, and one for the ones running in the background. Foreground and background processes were distinguished by their processing time. The process time for foreground processes must not exceed 500 ms.

Zhou used a more complex hardware model than SimSim. It models both process queues and allows multi-tasking. He points out that *"While the CPU scheduling policies in computer systems are usually more complicated, we feel that the above policy captures their essential features [...]"*[108, p. 1329]. Moreover, disk access was modelled as well. It was assumed that all disk accesses are handled by file servers, and that the access costs to them are equal for each processor. Finally, the time it takes to transfer a process and to calculate the load of a processor were, among others, estimated by empirical data. Now, Zhou could observe each algorithm's performance, its sensitivity to certain parameters, and other important features, such as scalability. For example, he found that the mean response time for the simulated processes, which is the time between the start and the end of a process, is indeed reduced by using many processors, but that the reduction stagnates when using more than about 30.

Soklic also used modelling and simulation to evaluate load balancing algorithms [90], but with a focus on load balancing algorithms for *client/server* architectures. This means, the load balancing algorithms manage objects on a set of servers, which are generated, destroyed and used according to client requests. Soklic used stochastic elements to model client behaviour, and defined additional parameters for load balancing algorithms. Then, he simulated the performance of three algorithms in different environments and analysed the results.

Both approaches use simulation techniques to evaluate load balancing algorithms, but they differ from the approach presented here in several aspects. Above all, they focus on load balancing algorithms in other problem domains: While Zhou's model is used to evaluate load balancing algorithms for *general* distributed systems, Soklic evaluates load balancing algorithms in the context of client/server architectures. Although we could regard a client/server software as a certain kind of simulation system, so that the client's model is equivalent to the application model and the system model is represented by a load balancing algorithm, not all simulation systems might fit into this scheme. In contrast to these approaches, SimSim is built to simulate arbitrary *PDES* systems, so it focuses on another application area.

Additionally, neither Zhou nor Soklic separate commonly used model parts from parts that are specific to their experiments[1]. Therefore, their approaches are hard to re-use. Finally, both approaches focus on load balancing issues, which is just one class of algorithms to be assessed by using SimSim.

In section 2.3, some load balancing approaches for PDES systems have already been mentioned. In [73], Oguara et al. introduce a load management algorithm for PDES-MAS. It moves SSVs in order to minimise the overall access costs of ALPs. The 'direction' from which an SSV is accessed by an ALP can be determined by the port through which the ALP's query reaches the CLP hosting the SSV. In the context of PDES-MAS, a port is a CLP's interface to a neighbour CLP or ALP. It is possible to count the number of SSV accesses for each port. A load management procedure is called periodically, which in principle moves each SSV to the CLP behind the port with the largest access count for this variable. Of course, the concrete algorithm is much more sophisticated, and uses various cost measures to decide whether to load balance, and which SSVs are eligible for load balancing at all.

Wilson et al. described and evaluated load balancing algorithms for SPEEDES (see section 2.3.3), but they focused on algorithms that employ a post-mortem analysis of a former run [103]. So, based on the definition of load balancing as described above, these algorithms are actually *partitioning* algorithms which use empirical data. An approach to balance the load of Time Warp simulations was presented by Carothers et al. [17]. In contrast to approaches that re-distribute model entities, their algorithm allocates logical processes to physical processors.

Of course, there are numerous other approaches, tailored to their specific applications (e.g. fractal approaches for particle simulations [11]). But, owing to the very specific requirements the abstract PDEVS simulator imposes on load balancing solutions, a more detailed discussion of these approaches will not yield better understanding of the problem to be solved in this chapter, and is therefore left out.

[1]Or, at least, they do not point this out in [108, 90]

5.2 Requirements Identification

Prior to developing a new load balancing algorithm, it is important to specify the requirements it has to fulfil. As section 5.1 illustrated, load balancing is a commonly (mis-)used term. Load balancing algorithms might be used to manage operating system processes [108], incoming requests for a set of servers [90], or model entities in a computer simulation [73].

So, what will be load balanced in case of the abstract PDEVS simulator? Evidently, the behaviour of a PDEVS model is defined by all of its atomic models, as well as the couplings among them. Coupled PDEVS models can be viewed as a help to re-use existing models. They encapsulate interrelated models and thus hide complexity from higher levels. They do neither have an own state, nor can their behaviour be customised (except for defining couplings).

Any computing load that might be introduced by the model is defined by the *atomic* models. Consequently, the load balancing algorithm should manage atomic models. Since each simulator component is coupled to exactly one atomic model, the simulators will have to be managed by the algorithm as well.

In addition to the computational costs that are introduced by the model, it might be important to consider the cost of handling coupled models. A coupled model and its coordinator should be hosted on the same processor as the atomic models in their subtree (and their simulator components), because otherwise additional communication cost would occur. All in all, the load balancing algorithm has to re-distribute model entities *and* the components to simulate them.

Although all elements the algorithm has to work with are clear by now, it is also necessary to consider the circumstances in which a load balancing algorithm is needed and has to perform well. Load balancing is invoked to account for *dynamic* changes of the model behaviour, which require a re-distribution to ensure an efficient parallel execution. This presumes that the changes of model behaviour are measured accurately, which is not necessarily trivial, especially when considering an environment in which *external* applications are executed too. For example, it is challenging to measure the exact load a logical process inflicts on a processor if the processor is shared by many LPs.

Due to external applications possibly running concurrently, the algorithm also has to keep track of certain hardware properties. While a processor's *overall* computing capacity is not likely to change during run-time, its *current* workload is affected by *all* applications it executes. At the same time, inter-processor communication capacities may also be used by external applications, so that it is important to measure them as well. In summary, a load balancing algorithm requires information about current processor workload and current inter-processor communication capacity on one hand, and the current computation and communication requirements of the PDES system on the other.

Now, the runtime information, which in principle defines the *present* state of the PDES system and its environment, has to be analysed. In order to re-distribute the PDES system to run more efficiently *in the future*, the load balancing has to *predict* its future behaviour on the basis of present (and past) data. Usually, it is quite optimistically assumed that the present state of a PDES system and its environment remains the same, so that the algorithm optimises the execution with respect to the current state. Alternatively, it would also be plausible to extrapolate the future behaviour from a time series of (past and present) runtime information. After the load balancing algorithm has decided which part of the PDES system to move, the migration has to be executed and the PDES system has to be re-configured accordingly (e.g. references have to be updated).

Finally, it is important to determine the frequency at which the load balancing will be executed. Since each execution of the load balancing algorithm means *additional* computing load and is *not* essential for the actual simulation, it may even hamper the overall simulation performance when called too often. On the other hand, a load balancing algorithm that is called seldom might not be able to speed up the execution at all (simply because it has no chance to react to sudden changes in the model's behaviour). Therefore, the frequency of load balancing execution has to be adapted to the volatility of the runtime information, which in turn depends on the dynamics of the PDES system and its environment.

In a more concise form, we can define the problems, for which a load balancing algorithm regarding the abstract PDEVS simulator in JAMES II has to provide suitable mechanisms, as follows:

- Measurement of run-time information

- Accumulation of run-time information
- Generation of a better placement for models, simulators, and coordinators
- Adaptation to PDES system and environment dynamics
- Migration of model entities *and* simulation components

5.3 A Benchmark Model

The load balancing algorithm will be developed using SIMSIM, which implies that its execution will be simulated at first. This motivates the development of an application model that can be used to control the extent to which the algorithm fulfils the requirements as framed in section 5.2.

Thus, the *benchmark model* that is detailed in this section can be seen as a generalisation of all application models, regarding their properties that have an impact on load balancing. Given that the relation between an application model's properties and the performance of load balancing algorithms is clear, it is possible to select the most suitable load balancing algorithm for any model whose properties are known.

Instead of developing a benchmark model, one could also use a real-world application model for testing. However, since the model is likely to be complex (see section 3.4) and therefore hardly mathematically tractable, some of its properties might be difficult to calculate. This is problematic, because the model's properties have to be easily controllable, otherwise a detailed analysis of the algorithm, e.g. the identification of sensitive model properties, would be impossible.

In [9, 10], Balakrishnan et al. present a framework to automatically generate benchmark models for arbitrary PDES systems. They motivate their efforts by claiming that *"The development of efficient parallel discrete event simulators is hampered by the large number of interrelated factors affecting performance"*[10, p. 1]. So, they address the same problems as described in section 2.3.3, but instead of *predicting* simulation performance to increase comparability among PDES systems, they propose a framework to generate benchmark models for *existing* PDES systems. Their performance analysis framework is able to generate benchmark models from definitions in the *workload specification language* (WSL). WSL allows the specification of *synthetic* simulation objects, which in turn represent model entities. Each WSL definition of a simulation object defines its key properties regarding execution, for example the size of its state, the granularity (i.e. how much computing time it needs to process a received event), or random number distributions that determine its output behaviour. All simulation objects are connected via edges and form a graph. Now, it is possible to implement components which transform this generic benchmark model to a benchmark model that is executable on a specific PDES system. By this procedure, different PDES systems can be compared with each other.

However, the performance analysis framework - although being extensible - does not yet support the definition of *dynamic* model behaviour, i.e. model behaviour that changes over time. This hampers its applicability in this context, since load balancing is *only* useful if the model behaviour is indeed dynamic. If a model's behaviour is not dynamic, a good partitioning algorithm or a single load balancing invocation would suffice to distribute the simulation properly. Nevertheless, the decisive parameters for PDES system performance (i.e. granularity, number of events, topology of the model, etc.) have already been identified by Balakrishnan and others [9]. These considerations should be reflected in the development of a dynamic benchmark model for the abstract PDEVS simulator.

Furthermore, Glinsky et al. propose a class of benchmark models for the DEVS formalism [40]. Similarly to the approach of Balakrishnan et al., they generate synthetic DEVS models to be executed. This means, they introduce three different structures of a coupled model, which then can be combined to form a model tree of width w and depth d. Each coupled model consists of $w-1$ atomic models and yet another coupled model, except the last coupled model in this chain, which simply consists of one atomic model. The depth of the tree is defined by the number of coupled models that are nested this way. The three types of coupled models only vary in the definition of their submodel's couplings. As parameters for atomic models, Glinsky et al. define different execution times for δ_{int} and δ_{ext}.

This approach suffers from several shortcomings: Firstly, Glinsky et al. do not define the *complete* behaviour of an atomic model, since they do not give parameters for their time advance functions[2]. Secondly, the tree structure of the benchmark model is strictly limited. It is not possible to create a benchmark model where a coupled model consists of *multiple* coupled models. This is an artificial limitation that hampers the representativeness of the benchmark model. Moreover, *individual* behaviour of atomic models cannot be modelled, because general execution times for δ_{int} and δ_{ext} are used. Besides that, the extent of communication, e.g. the size of the input and output data they exchange, is left out of the benchmark model. Finally, this model is also incapable of changing its behaviour dynamically.

All in all, both approaches for synthetic models have serious shortcomings, and particularly their disability to change their behaviour at runtime makes them ineligible in this context. Of course, there *are* models that show dynamic behaviour and are even commonly used as benchmark models (e.g. Tileworld [80]). But these models are quite complex, and their implementation might be tedious. Besides that, their behaviour is not easily parameterisable. This means, the *extent* of dynamism regarding computing load or communication cannot be defined by adjusting a single parameter. Rather, the exact load and communication requirements of these benchmark models can be hard to predict, whereas a suitable benchmark model should possess means to control all of its relevant properties easily.

Therefore, a new benchmark model to test the load balancing algorithm was worked out and implemented for SIMSIM's JAMES II plug-in and for JAMES II itself. Any tree can be defined to be the model tree of the benchmark model. Usually, these trees are generated randomly, by giving the number of nodes and the average branch factor (i.e. the average number of children per node in the tree).

Now, a unique ID is assigned to each model, beginning at the topmost coupled model. It gets the ID 0, and all of its submodels get the IDs $1 \dots k$. Then, all submodels of these submodels are processed. They get the IDs $k \dots j$, beginning with the submodels of the submodel having the ID 1, and so on. The node for the root coordinator is left out, but the topmost coupled model's parent ID is set to -1, so that it can be identified easily. This numbering is illustrated in figure 5.1.

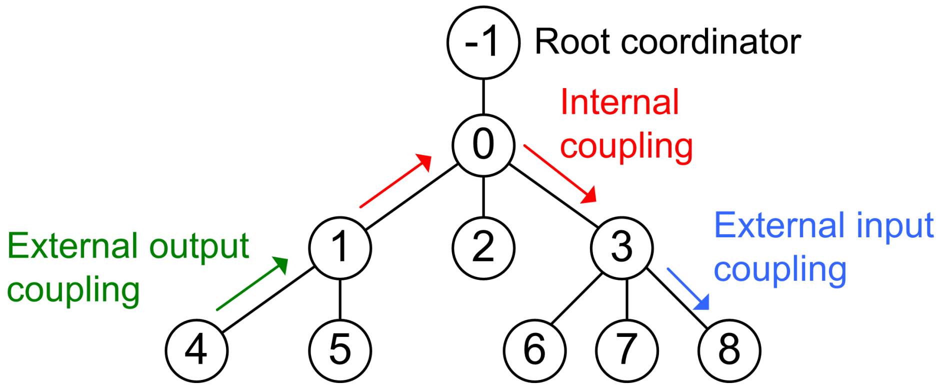


Figure 5.1: Sample Benchmark Model Tree

To make the benchmark model as flexible as possible, each atomic model is 'directly' coupled with each other atomic model in two directions, which means that there are additional couplings in all coupled models between them. Thus, an atomic model can directly send an output to any other one. This leads to numerous couplings, but the number of couplings is not necessarily decisive for the performance of a coupled model. A coupled model still has to handle a limited amount of models, and a coupling will only be used when two specified atomic models communicate with each other in *one* direction. Figure 5.1 also illustrates how two arbitrary atomic models can be connected. Since the abstract PDEVS simulator in JAMES II minimises communication cost (see section 2.3.1), this will *not* result in more messages. However, the execution of the abstract PDEVS simulator components might be slightly hampered, because each one has

[2]At least, they do not so in [40]

to handle many couplings.

The key problem is now to define the behaviour of the atomic models (which will just be called models in the following paragraphs), since this is now the determining factor of the overall benchmark model's behaviour. To support dynamic behaviour and to facilitate its definition, each atomic model is parameterised with a list of *state phases* and a number of *rounds*. A state phase completely defines the behaviour of an atomic model for a certain duration in application model time, which is part of its definition. After the duration of one state phase has expired, the model's behaviour is defined by the next state phase in the list. The state phase that defines the model's behaviour at a certain point in time is named the *current state phase* of the model. When the end of the list has been reached, one round is finished and the same procedure starts over. The round counter is incremented. If the number of rounds to be executed is reached, the atomic model just behaves 'passively'; it does not generate any new messages. If a message is received, the settings of the last state phase apply. Using this approach, it is possible to define individual 'cycles' of dynamic behaviour without much effort. Each state phase consists of various parameters that are best understood when the semantics of the benchmark model are clear.

The atomic models are regarded as individual entities that exchange messages, so this benchmark model can be seen as a modified PHOLD model (see glossary). When a message arrives at its destination, it will reside there for a random application model time, which is defined by the distribution $\Delta_{reactionT}$ of the recipient's current state phase. Afterwards, it will be sent back to its sender. Each model generates new messages continuously. How much messages a model will generate during a state phase is determined by $\Delta_{nextMsgT}$, which is the distribution of times between generated messages. So, if $\Delta_{nextMsgT} = N(1, 0)$, i.e. a normal distribution with mean one and no variance, the model will generate a new message for each time unit of application model time that has passed. This behaviour continues for the duration of its current state phase. The size of a generated message is random and defined by $\Delta_{msgSize}$. With these distributions, one is able to define the extent of communication among the models. Each model can change its communication behaviour arbitrarily.

Besides that, it is desirable to control the way each model selects its communication partners. This is particularly interesting for testing the abstract PDEVS simulator, because the sending of a message causes its propagation to the destination (via the couplings), and thereafter *both* sender and receiver will execute their state transition *in parallel*. So, sending a message from model A to model B results in a (potentially) parallel state transition of A and B.

To model the computational effort a model's state transition functions may require, the distributions $\Delta_{\delta_{int}T}$ and $\Delta_{\delta_{ext}T}$ have to be defined in each state phase. Until now, these distributions represent the simulated wallclock time each model needs for the processing. This facilitates the experiments, as described in chapter 6. Since the execution of δ_{con} means that a model has received *and* sent a message simultaneously, the time to execute a confluent state transition is set to $\Delta_{\delta_{int}T} + \Delta_{\delta_{ext}T}$.

In principle, the recipient selection for a newly generated message follows a uniform distribution. Without any other mechanism, this would result in a uniform distribution of parallel executions as well. Clearly, this model would not suffice to test whether the algorithm is able to cope with unevenly distributed parallelism. To allow arbitrary degrees of parallelism between models, these are organised in *groups*. For each state phase, a so-called *preference factor* $pref \in [0, 1]$ defines the extent to which group members are preferred over other models as recipients. Let n_g be the number of other models in a model's group[3], and n_m the number of all other atomic models. When there are any models to send a message to, i.e. $n_m > 0$, the probabilities

$$P_{chooseFromCluster} = \frac{n_g + pref \cdot (n_m - n_g)}{n_m} \tag{5.1}$$

$$P_{chooseFromRest} = \frac{(1 - pref) \cdot (n_m - n_g)}{n_m} \tag{5.2}$$

determine how much a model prefers to send a message to a member of its group. Note that $pref$ serves for a simple linear interpolation, and that $P_{chooseFromCluster} + P_{chooseFromRest} = 1$.

[3]A model cannot send a message to itself

Name	**Use**
$name$	Name of the state phase
$timeSpan$	Duration of the state phase
$\Delta_{nextMsgT}$	Distribution of time between generated messages
$\Delta_{msgSize}$	Distribution of message sizes
$\Delta_{\delta_{int}T}$	Distribution of computational load for δ_{int}
$\Delta_{\delta_{ext}T}$	Distribution of computational load for δ_{ext}
$\Delta_{reactionT}$	Distribution of reaction time for received messages
P_{change}	Probability of changing the group after the state phase has ended
$pref$	Preference factor

Table 5.1: Parameters of a State Phase: An additional name field helps the user to explain the meaning of a state phase. $\Delta_{\delta_{int}T}$ and $\Delta_{\delta_{ext}T}$ represent simulated wallclock time (wallclock time in the original system), whereas $\Delta_{nextMsgT}$ and $\Delta_{reactionT}$ refer to application model time (simulation time in the original system).

Finally, this model grouping should be dynamic as well. Therefore, a last parameter for the state phase is introduced. P_{change} defines the probability that a model changes its group after the current state phase is over. To change its group, a model has to send a request to the *group manager*, which is the model with the smallest ID. The group manager model behaves just as any other model, except that it processes requests for re-grouping. This had to be centralised, because a decentralised protocol to switch groups would have to cope with many special cases, and thus would be hard to develop[4]. The group manager processes all re-grouping requests and immediately (i.e. by setting $ta = 0$) sends messages containing the updated neighbour lists to all models that need to be notified. Table 5.1 summarises all parameters of a state phase.

This model is simple enough to be mathematically tractable, but is still able to express dynamic changes of a model's sending preferences, calculation costs, and communication costs. Indirectly, this allows to test the algorithm for differently grained models and models with various degrees of inherent parallelism.

To facilitate the control of parallelism, a *rounding uniform distribution* U_r was implemented, which rounds uniformly distributed random numbers to k values within a range. The values are placed by a recursive function that puts one value in the middle of the interval and then applies itself to place $\lceil \frac{k-1}{2} \rceil$ values in the lower half and $\lfloor \frac{k-1}{2} \rfloor$ values in the upper half. Hence, for $k \to \infty$, this distribution is approximating the uniform distribution, but the smaller k is, the higher the chance that, for example, two models use the *same* point in time to send a message. Thus, the smaller the k, the higher the chance that the activities of the models can be processed in parallel.

The benchmark model was implemented for JAMES II at first, and was then ported to SIMSIM's JAMES II plug-in. Again, the lines of code were counted to roughly compare the implementation efforts. Although re-used components like the `StatePhase` class were excluded, the complexity of the real-world benchmark model was much higher: Its implementation consists of ≈ 900 lines of code, whereas the implementation of the application model for SIMSIM's JAMES II plug-in just took ≈ 450 lines of code. Afterwards, both models were tested for equivalent behaviour, given that they are configured similarly (see CD).

5.4 Development of a Prototype Algorithm

This section describes the development of a prototype load balancing algorithm, aided by the use of SIMSIM. SIMSIM was used to test new ideas, and to identify flaws in design and programming as early as possible.

Since the prototype will work on the system model rather than the real system, it is possible to select

[4]Consider two models trying to swap their group membership at the same time. How can each of them figure out which new neighbours it has? An asynchronous protocol was desired, so that the biasing communication overhead is minimised.

the most important requirements from the ones defined in section 5.2, and assume that the others can be solved later. By assuming that the load balancing algorithm is able to measure communication and computation costs of each PDEVS model accurately, the task of developing the actual load balancing algorithm is greatly facilitated. Therefore, the way the load balancing algorithm accumulates information will still be discussed, but concrete algorithms to obtain this data will be left out. Moreover, the migration of models, simulators and coordinators will be done by simply updating their position data, so there is no *real* migration from one machine to the other. In spite of this, we can assume that it *is* possible to migrate these components, so that these mechanisms can be developed later.

All in all, only three of five requirements stated in section 5.2 need to be considered in this prototyping phase. For load balancing algorithms in general, Zhou identifies three components: *information policy*, *transfer policy*, and *placement policy* [108, p. 1330]. The information policy defines the information needed by the algorithm, and how it is managed. This is clearly one of the requirements (Collection of run-time information), and has also to be fulfilled by the prototype. Transfer and placement policy define which elements are eligible for load balancing, and how new locations are found for them, respectively. Both policies are reflected in the requirement to generate a better arrangement of models, simulators and coordinators. Additionally, it is required to let the load balancing algorithm adapt to the dynamism of simulation demands.

Now, with these three requirements left, we can still narrow down possible solutions. To develop a suitable information policy, it has to be decided whether the load balancing algorithm should work centralised or decentralised. For the specific problem of load balancing a PDEVS simulation, it seems advantageous to implement a centralised load balancing algorithm. Centralised algorithms are usually easier to develop and debug, because all activity is handled by a single component. Moreover, the PDEVS simulation works in a hierarchical manner, so that load balancing on the processor hosting the root coordinator seems natural. This facilitates the collection of load balancing information, which can be exchanged along the edges of the abstract simulator tree.

Furthermore, it can be assumed, without the loss of generality, that root coordinator and topmost coordinator will *always* be hosted on the same processor. Since all other parts of the simulation can be arbitrarily moved, this does not reduce the number of possible load balancing mappings too much. This assumption will facilitate the execution of the algorithm in JAMES II, since the root coordinator, which will trigger the load balancing, will not have to migrate itself. The root coordinator is solely associated with the topmost coordinator, so that both components should be placed on the same host.

Finally, we can also reduce the number of elements that have to be considered for migration. Since each simulator and coordinator is directly coupled to *one* element of the model tree, it is sufficient to only consider the models they handle. Each coordinator and simulator should be placed on the processor that hosts its PDEVS model.

Another issue that needs some consideration is the compatibility of the load balancing mechanism to the DEVS partitioning algorithm described in [29]. Of course, both algorithms are compatible in principle, but the DEVS partitioning algorithm supports the definition of *constraints* to specify further requirements. For example, one could use constraints to statically assign a model to a certain processor. By this means, it is possible to set the location of models that have interfaces to external processes (e.g. agents) beforehand. However, the support for constraints can be quite complex, so it is left out in the context of this prototype implementation.

As already described in section 2.1.2, only events that occur at the same point in simulation time can be parallelised when simulating a PDEVS model using the abstract PDEVS simulator. In [99], Troccoli et al. assume that all *-messages cause the same amount of computational load, so that they conclude: *"If all logical processes received the same number of (*,t) messages, then the load is evenly distributed"*[99, p. 72]. While this might be a valid assumption in their context, it is clearly invalid in others. Indeed, the assumption that each calculation takes the same amount of time is quite unrealistic. Consider two atomic models that make use of very complex computations, and a number of simple atomic models. When we assume that both complex models can often be executed in parallel (i.e. their state transition functions are often activated at the same points in simulation time), it would clearly be best to host these models on different processors [100]. Just counting the number of *-messages will not identify this interrelation. So, a load balancing algorithm that bases on this assumption might place both models on the same logical processor, as long as the number of *-messages is evenly distributed. Hence, both execution time *and* *-messages have to be taken into account by a load balancing algorithm.

The restriction to only exploit inherent parallelism and the dependence on the calculation time of atomic models demand a re-definition of *load* in the context of PDEVS models. Of course, the load of a processor could simply be defined as the time that it took to simulate all model parts it hosts. But if all of these executions had to be done *sequentially*, e.g. because they occurred at different points in simulation time, the performance of the simulation will *not* increase when re-distributing the elements that induced this kind of load. In this case, speedup can only be achieved by assigning *all* of these model elements to a faster processor.

To define the load that *should* be balanced to achieve simulation speedup, we have to consider all operations that one processor executes, but that could also be executed *in parallel* instead. Even though this definition of load is suitable in that it correctly identifies the elements whose migration would be beneficial, it still does not conform to the common meaning of load in PDES systems. Usually, one would regard load as a static factor that has to be distributed evenly. Following the definition of load as proposed here, the overall goal of the algorithm is to *minimise* load (i.e. the amount of parallel executions that are done sequentially). So, it has to be concluded that the meaning of load in this context strongly differs from the commonly used term.

Another important factor is the communication delay, also called communication cost, that occurs when a message is passed from one physical processor to another. Although the abstract PDEVS simulator of James II minimises the *amount* of messages between elements of the abstract simulator tree, these messages still have to be propagated to the imminent models, etc..

How can these communication costs be integrated with load balancing reasoning? In the case of PDEVS, one processor (i.e. the coordinator that is hosted on it) might send a *- or an x-message to a second processor that hosts one of its sub-models. The second processor will then progress by possibly propagating messages to other processors and executing either the λ or the state transition functions of the atomic models it hosts. Then, a y-message or a done-message is returned to the first processor. The communication overhead that can occur depends on the computational load of the first processor during the execution of the second one. If the first processor is longer busy than the sum of the second processor's execution time and communication delay, the communication overhead is completely *hidden*. This technique of *hiding communication cost* is used quite often to optimise parallel computations (e.g. see [4]). Figure 5.2 shows the difference between hidden and unhidden communication delay d. It implies that the extent to which loads are migrated to additional processors has to be evaluated carefully.

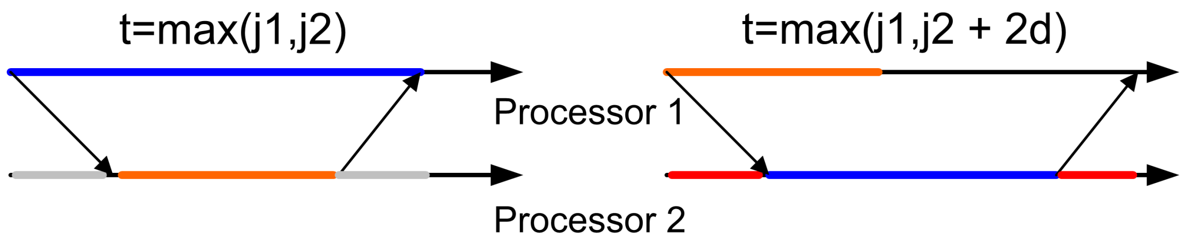


Figure 5.2: Hiding Communication Delay. On the left side, the communication delay d does *not* need to be considered for the calculation of the overall execution time t, because it is entirely 'hidden' by the parallel execution of a job on processor one (blue). On the right side, the parallel job execution on processor one (orange) is shorter than the execution time of processor two, and thus not able to hide the communication delay (red).

To conclude these preliminary considerations, the aforementioned circumstances should be expressed in a more concise, mathematical way. Consider two atomic models, m_1 and m_2, which are associated with the same coupled model. Without loss of generality, let the computation time of model one be lesser than the time it takes to compute model two, i.e. $c_{m_1} \leq c_{m_2}$. Moreover, let $c_{m_1} > 0$, i.e. there *is* some computation effort to execute the models.

Now, assume that m_1 and m_2 both are imminent models. Then, the coordinator of the coupled model has to exchange four messages with both of them for this simulation step. It sends a *-message to m_1 and m_2, receives their y-messages, sends x-messages to them, and finally receives their done messages. Let the message transfer time be negligible on a single machine, so that it takes $c_{m_1} + c_{m_2}$ time to execute both models on a single host. In a parallel setup, we would rather migrate the child causing a *smaller* amount of computing time, which is m_1 in this case. This is done to hide the communication cost. So, in a parallel setup with an average message transfer time d, the execution time would be $\max(c_{m_1} + 4 \cdot d, c_{m_2})$. We are interested in the case where parallel execution is advantageous, so that it takes less time:

$$\max(c_{m_1} + 4 \cdot d, c_{m_2}) < c_{m_1} + c_{m_2} \tag{5.3}$$

Inequation (5.3) can be analysed using case differentiation. If $c_{m_1} + 4 \cdot d \leq c_{m_2}$, it is reasonable to migrate m_1 to another processor, since inequation (5.3) can be written as $c_{m_2} < c_{m_1} + c_{m_2}$, which is true (it was assumed that $c_{m_1} > 0$). If $c_{m_1} + 4 \cdot d > c_{m_2}$ instead, inequation (5.3) can be simplified to $c_{m_1} + 4 \cdot d < c_{m_1} + c_{m_2}$, which is true if $4 \cdot d < c_{m_2}$.

Hence, it is sufficient to test whether $c_{m_2} \leq 4 \cdot d$ holds true. If it does, the second case is true ($c_{m_1} + 4 \cdot d > c_{m_2}$) *and* $4 \cdot d < c_{m_2}$ is false. This is the *sole* case in which a migration of m_1 is *not* benificial.

5.4.1 Version 1: Identifying Parallelism

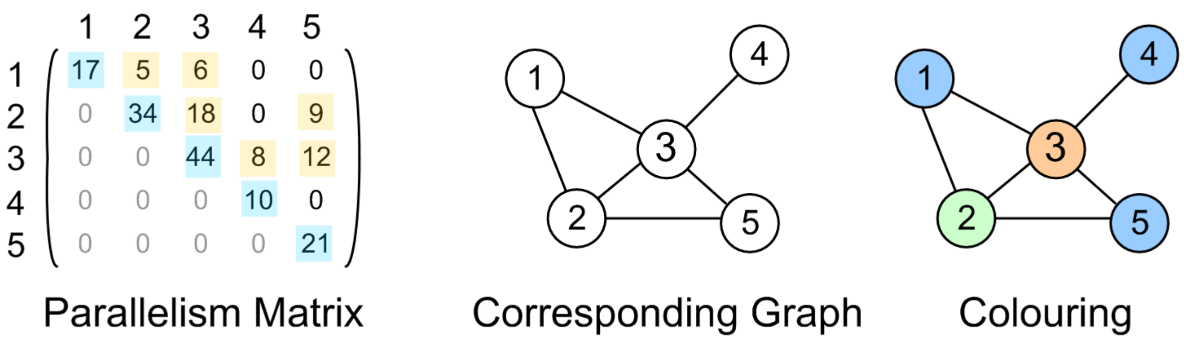
The first prototype of the algorithm was built around its ability to identify parallelism, because inherent parallelism is, as described before, the main factor to be considered for load balancing of PDEVS models. Since each coordinator hides its sub-models from its own coordinator, it can only log the parallel execution of its sub-models by itself.

This was realised by equipping each coordinator with an analysis component that extends the class `AbstractLoadBalancingInformationAnalyser`. Thus, each load balancing algorithm can define its own information policy (again, the strategy design pattern was used [38, p. 315 – 323]). The analyser is called for all messages a coordinator sends. When the load balancing algorithm is started, it has to be initialised first. The abstract simulator tree is traversed and all load balancing information that was gathered by its analyser components is accumulated. The new type of *lb-message* that is used to do so is handled by the analyser class as well. When the initialisation is finished, the load balancing algorithm is called. This realisation of an extensible information policy was tested using SimSim. Now that it is clear which component is in charge of collecting the runtime information, and how this runtime information will eventually reach the load balancing algorithm, it still has to be defined *which* data ought to be collected.

Basically, any combination of sub-models may be imminent at the same time. These combinations of imminent sub-models can vary from simulation step to simulation step. Moreover, some sub-models that were not imminent might receive inputs from other imminent models in the model tree, and are therefore executing their state transitions as well. In the following, we assume that the execution of λ, which is an atomic model's output function, will introduce negligible load. This assumption is fair, because λ does not change the state of the model, but simply generates output that corresponds to the model's current state (see section 2.1.1). This implies that all models which receive either a *-message or an x-message in one simulation step can execute their (possibly time-consuming) state transition functions in parallel, and therefore have to be logged.

Having said that, it is clear that storing a list for each of the simulation steps, which had to contain the IDs of all models that executed a state transition at this point, would result in a huge memory overhead. To avoid this, a *parallelism matrix* $P = 0^{n \times n}$ is introduced. It is generated for each coordinator, n being the number of its coupled model's sub-models. Each model m is associated with an index i. Since all sub-models have unique integer IDs, there is a total ordering between them. The model with the smallest ID has the index 1 and is denoted as m_1, the model with the next-larger ID has the index 2, and so on.

Now, we can use P to keep track of the *binary* parallelism relation between sub-models. Each time a model m_i executes a state transition function in the same simulation step as model m_j and $j > i$, the parallelism between m_i and m_j is logged[5] by incrementing $P_{i,j}$. Additionally, each time a model m_i executes a state transition function, $P_{i,i}$ is incremented to count the overall state transitions of m_i.



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 17 & 5 & 6 & 0 & 0 \\ 2 & 0 & 34 & 18 & 0 & 9 \\ 3 & 0 & 0 & 44 & 8 & 12 \\ 4 & 0 & 0 & 0 & 10 & 0 \\ 5 & 0 & 0 & 0 & 0 & 21 \end{array}$$

Figure 5.3: Interpretation of Parallelism Data. The parallelism matrix for the sub-models $m_1, \ldots, m_5$ can be transformed to the adjacency matrix of a simple, undirected graph. Graph colouring can be used to identify sets of models that ought to run in parallel, i.e. on different hosts.

A sample parallelism matrix is shown in figure 5.3. Due to its definition, it is an upper triangular matrix, which stores each sub-model's total number of state transitions on the main diagonal, and each simultaneous execution in the fields above the main diagonal. When the collection of load balancing information has finished, all fields in the matrix are re-set to zero.

The advantage of the parallelism matrix is, that *each* parallel execution will be registered at some coordinator in the abstract simulator tree. If two models which are *not* managed by the same coordinator are executed in parallel, this will be noted by the coordinator that coordinates the minimal sub-tree that includes both models.

How can parallelism matrices now be used to find elements of the model tree that should be migrated? Basically, model elements that are executed in parallel should be placed on different hosts, and thus model elements that do *not* run in parallel can be placed on the same host. Since parallelism matrices only record the amount of parallelism for each *pair* of a coordinator's sub-models, a way to identify more complex interrelations, e.g. the frequent parallel execution of *three* models, has to be found.

At first, the parallelism matrix could be used to generate a graph of all sub-models. As graphs are in principle a representation of a set and a binary relation between its elements, this formalism suits well for further processing the parallelism information. Each sub-model is represented by a node in this *parallelism graph* G_P (see figure 5.3). An undirected edge is added between the nodes of all models m_i, m_j $(i < j)$ that have been executed in parallel, which is the case if $P_{i,j} > 0$. In other words, P can be transformed to an adjacency matrix of a simple, undirected graph G_P, by setting all values in the main diagonal to zero and setting $P_{j,i} = P_{i,j}$ for $i < j$.

Even though the parallelism information is now described more expressively, it is still not clear how to determine the sets of submodels which should be migrated. A mechanism to generate these sets would have to place two models in a different set, if their corresponding nodes in G_P are neighbours. Moreover, it is important to minimise the number of used processors, i.e. the number of submodel sets. This problem is equivalent to the *graph colouring problem*.

Graph colouring is a classic problem in graph theory. The nodes of a simple, undirected graph are coloured, so that each node has a colour that differs from its neighbour's colours. The graph colouring problem is to find the *minimal* number of colours that are needed to colourise an arbitrary graph, which

[5]The condition $j > i$ ensures that P is an upper triangular matrix (see figure 5.3). It accounts for the fact that the parallelism matrix would be symmetric otherwise, and thus would contain redundant information.

in this context corresponds to the minimal number of processors to distribute the sub-models on. Unfortunately, graph colouring is an NP-hard problem and the algorithms to solve it are still too costly to work on large graphs [28].

Nevertheless, one could use a simple *depth-first search* to colourise the graph and *approximate* the minimal number of used processors. Afterwards, all models whose nodes have been coloured with the same colour can be placed on a single host, because they were not executed in parallel. Now, the parallelism matrices of the upper coordinators could be analysed this way, and the identified sets could be placed on available processors.

This approach has been developed and tested using SIMSIM. It was able to identify the parallelism that occurred. The next section describes how the preliminary considerations regarding the benefits of distributed simulation can be integrated into this algorithm. This is necessary, because the granularity of the model might be too low to be distributedly simulated at all, which *cannot* be deduced by simply analysing the parallelism matrices.

5.4.2 Version 2: Integrating Computing Load

A more advanced version of the load balancing algorithm should be able to consider the load that the state transition functions impose. Without further information, it is impossible to decide which migrations yield the maximal gain. For example, the distribution of two complex models which were executed in parallel only once could *still* be better than distributing models which might have been called hundreds of times in parallel, but did not introduce any load.

To measure the load, each information analyser that is associated with a simulator stores the *overall* time the atomic model needs to execute its state transition functions. This approach implicitly presumes that all processors have identical computing power, otherwise the calculation time would have to be scaled by an additional factor (see section 5.5). A coordinator's information analyser solely adds up the execution times of its sub-models, so that communication delays are excluded from the execution time calculation. In conjunction with the overall *activity count* $a_i = P_{i,i}$, it is now possible to calculate the average calculation time c_i for any model m_i.

This allows to apply the preliminary considerations from page 72, so that the *gain* of using parallel instead of sequential simulation can be calculated. Consider two sub-models m_i and m_j $(i < j)$ that belong to the same coupled model. The gain of distributing m_i and m_j will only be calculated when $P_{i,j} > 0$, otherwise the models have not been executed simultaneously and thus their distribution cannot speedup the execution. The average calculation times of m_i and m_j are denoted as c_i and c_j. Their activity counts are defined as $a_i = P_{i,i}$ and $a_j = P_{j,j}$. Finally, an average communication delay $d > 0$ is assumed.

A time estimation for executing m_i and m_j on one host, i.e. sequentially, can now be written as:

$$a_i \cdot c_i + a_j \cdot c_j = (a_i - P_{i,j}) \cdot c_i + (a_j - P_{i,j}) \cdot c_j + P_{i,j} \cdot (c_i + c_j) \tag{5.4}$$

Recall that $P_{i,j}$ is the number of potentially parallel executions of m_i and m_j. The terms $(a_i - P_{i,j})$ and $(a_j - P_{i,j})$ define the number of m_i's executions when m_j was not executed, and vice versa. The execution time having m_j executed on another processor can be estimated by

$$(a_i - P_{i,j}) \cdot c_i + (a_j - P_{i,j}) \cdot (c_j + 4 \cdot d) + P_{i,j} \cdot \max(c_i, c_j + 4 \cdot d) \tag{5.5}$$

Since m_j is located on another processor, $4 \cdot d$ has to be added to its calculation time for the parallel executions (last summand), as well as to all of its $(a_j - P_{i,j})$ executions without m_i. Having estimations for both sequential and distributed simulation of m_i and m_j, the gain of migrating m_j can be estimated by subtracting the parallel estimation (5.5) from the sequential one (5.4):

$$\begin{aligned} &((a_i - P_{i,j}) \cdot c_i + (a_j - P_{i,j}) \cdot c_j + P_{i,j} \cdot (c_i + c_j)) - \\ &((a_i - P_{i,j}) \cdot c_i + (a_j - P_{i,j}) \cdot (c_j + 4 \cdot d) + P_{i,j} \cdot \max(c_i, c_j + 4 \cdot d)) \\ &= P_{i,j} \cdot (c_i + c_j) - (a_j - P_{i,j}) \cdot 4 \cdot d - P_{i,j} \cdot \max(c_i, c_j + 4 \cdot d) \end{aligned} \tag{5.6}$$

Now, the same case differentiation as for inequation (5.3) can be done for (5.6). If c_i is large enough to hide the additional communication costs, i.e. $\max(c_i, c_j + 4 \cdot d) = c_i$, the gain of migrating m_j can be estimated by using (5.6) as

$$\begin{aligned} &P_{i,j} \cdot (c_i + c_j) - (a_j - P_{i,j}) \cdot 4 \cdot d - P_{i,j} \cdot c_i \\ &\quad = P_{i,j} \cdot c_j - (a_j - P_{i,j}) \cdot 4 \cdot d \end{aligned} \tag{5.7}$$

This means, migrating m_j to another processor is advantageous if – and only if – the communication overhead for its $(a_j - P_{i,j})$ single activations is less than the parallelised computation load $P_{i,j} \cdot c_j$. In contrast, if $c_i < c_j + 4 \cdot d$, the gain is

$$\begin{aligned} &P_{i,j} \cdot (c_i + c_j) - (a_j - P_{i,j}) \cdot 4 \cdot d - P_{i,j} \cdot (c_j + 4 \cdot d) \\ &= P_{i,j} \cdot c_i + P_{i,j} \cdot c_j - a_j \cdot 4 \cdot d + P_{i,j} \cdot 4 \cdot d - P_{i,j} \cdot c_j - P_{i,j} \cdot 4 \cdot d \\ &= P_{i,j} \cdot c_i - a_j \cdot 4 \cdot d \end{aligned} \tag{5.8}$$

Note that the gain is indeed *negative* if $4 \cdot d > c_i$ ($P_{i,j} \leq a_j$ by definition), which was already identified as a critical condition during the preliminary considerations. In contrast to inequation (5.3), (5.7) and (5.8) do *not* assume $c_i > c_j$, but instead can be used to calculate the gain of migrating either one model or the other. This is important, because it allows to calculate the gain of migrating a *set* of sub-models using (5.7) and (5.8). The sub-model sets are generated by the graph colouring mechanism. Their migration eligibility can be compared by the deduced estimations.

At first, instances of an Edge class are used to decorate the edges of the parallelism graph. For each edge, the gains of distributing either of the neighbours is calculated. Then, it is also useful to filter out edges where both gains are negative, i.e. a distributed simulation would not be beneficial at all. By letting the graph colouring algorithm ignore these edges, it might need fewer colours (i.e. sets, processors) and could even colourise both neighbours with the same colour, thus avoiding the negative gain.

Afterwards, each set is analysed by considering *all* of its edges to other sets. The overall gain of a set is stored, together with the set, to a global list. This procedure continues for all coordinators. Finally, the list of gains is sorted, and the $p-1$ sets with the greatest gain, p being the number of available processors, will be distributed. This is done by traversing the tree and splitting it up where the sets with the greatest gain have been identified (see figure 5.4). If there are less than $p-1$ sets with gain, the remaining processors will not be used.

The model entities and their simulation components are already situated on remote hosts. Each migration of a model element costs time, because its host has to be notified. Then, the host processor migrates the model and its associated coordinator or simulator to the new location. This takes some time as well, depending on the size of the model's state and the network connection. Finally, references to the moved objects have to be updated throughout the system.

To minimise migration cost, a voting scheme was added to the algorithm. Let $u \leq p$ be the number of model sets to be executed on a single processor. In other words, u is the number of colours used to colourise the model tree in figure 5.4. All models that belong to the topmost coordinator's set will be moved to the topmost coordinator's host processor, as already described at the beginning of this chapter. So, one set is already mapped to a processor.

The migration matrix $M = 0^{(u-1)\times(p-1)}$ can now be used to find a good migration mapping that reduces the overall migration cost for the other sets. The host processor p_h is retrieved for each model, and $M_{s,h}$ is incremented. s is the index of the set the model is currently in.

When the calculation of the migration matrix is finished, the maximal value

$$max_{1,i,j} = \max_{1\leq i\leq u-1, 1\leq j\leq p-1} M_{i,j}$$

is chosen to determine the next element of the mapping. All models of set i are mapped to processor j, because the definition of $max_{1,i,j}$ ensures that no other set has more elements hosted by processor j, and

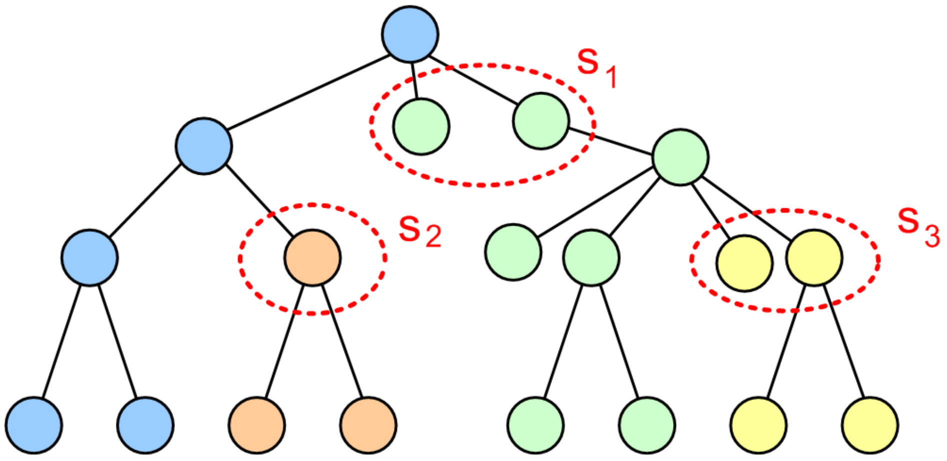


Figure 5.4: Splitting up the Model Tree: Migrating the identified sets s_1, s_2, and s_3 would result in a maximal estimated gain when the model behaviour remains the same. The colours of the nodes denote to which set (i.e. processor) they belong. The root coordinator is irrelevant in this context and therefore not depicted.

no other processor hosts more models of set i. Since the models of set i are now assigned to a processor, and processor j is not available for other sets any more, line i and column j are removed from M. This procedure can now be continued.

One can choose $max_{k,i,j} = \max_{1 \leq i \leq u-k, 1 \leq j \leq p-k} M_{i,j}$ for $k = 1, \ldots, u-1$ (by definition, $u \leq p$), each time assigning set i to processor j. When using this mapping, less than $max_{1,i,j}$ redundant migrations have to be done (see derivation in appendix A). Figure 5.5 shows the effect of this voting mechanism as predicted by SIMSIM. For these experiments, the default parameters were used. They are described in section 6.1.

It has to be recognised that this approach of generating a new placement of model entities does *not* consider different computing capacities, i.e. it implicitly assumes an environment of homogeneous systems. However, it can be implemented easily and should suffice for a prototype implementation of the algorithm. Later on, more sophisticated mechanisms could be added without too much trouble (see section 5.5). The pseudo code describing the general load balancing approach is given in algorithm 2.

Another aspect of the problem that was not yet integrated with the algorithm is the cost of model migration. As described before, this cost might be quite important. Therefore, an average migration cost for models is assumed. Consider two models having the same gain of being distributed, one is a coupled model with thousands of children and the other is an atomic model. To save migration costs, it is preferable to migrate the single model instead of the coupled one[6].

Hence, the gain calculation is adjusted to consider the overall size of the sub-tree a model handles, multiplied by the assumed average migration cost $migrationCost$. Using these parameters makes the algorithm migrate more often when the migration costs are low (e.g. in a fast network) and rather seldom when the migration costs are high (i.e. on a grid). The effect of this mechanism is compared to the basic algorithm in figure 5.5. Although the algorithm should work quite well by now, there is still one requirement left, which is the adaptation to the dynamism of the system (see section 5.2). This will be the focus of the next section.

[6]Of course, this is only true if the state sizes of the models do not differ too much. However, this consideration would complicate the algorithm quite a lot, so it was left out here.

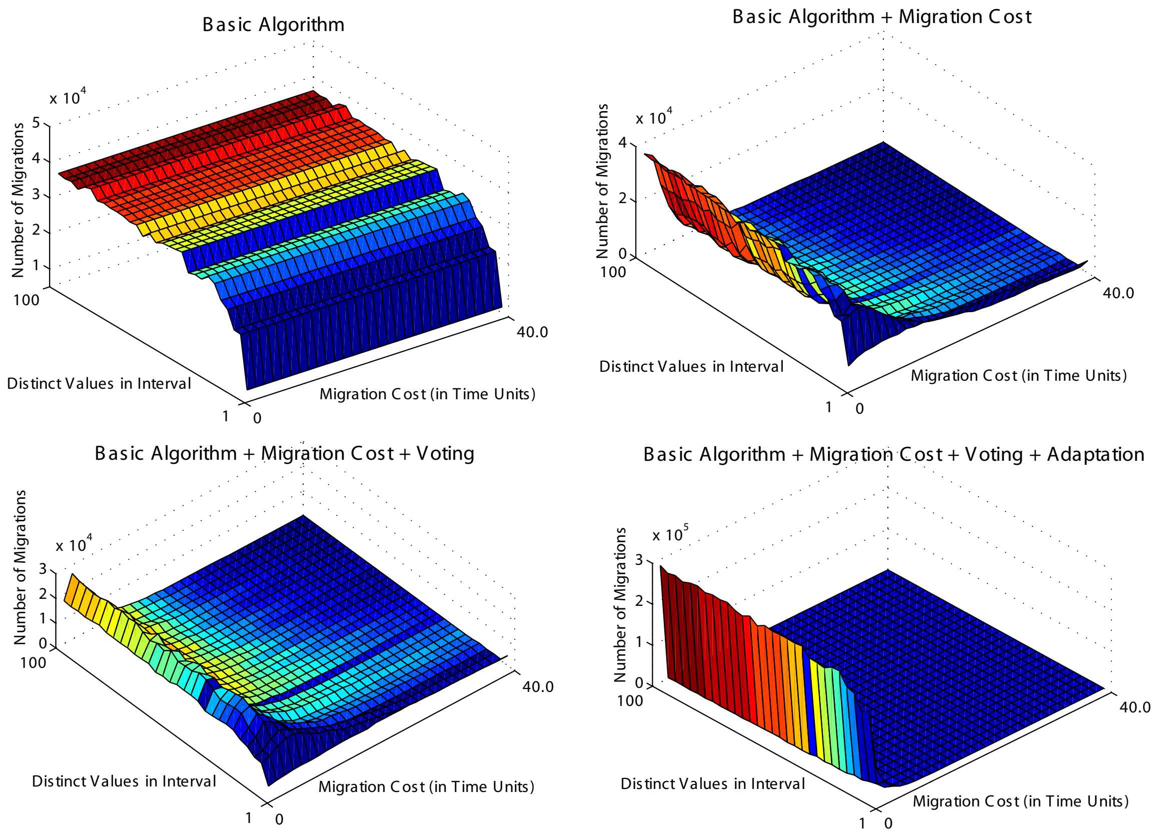


Figure 5.5: Number of Migrations when using Different Variants of the Algorithm. These plots show how the number of migrations is affected by migration cost and inherent parallelism, which is controlled by the number of distinct values in the rounding distribution of message send times: the higher the number, the lesser the parallelism. Note the abnormal behaviour of the algorithm equipped with the adaptivity mechanism (lower right corner). Its migration number is one order of magnitude larger under certain circumstances. It can also be seen that the consideration of migration cost (upper right corner) and the voting scheme (lower left corner) reduce the number of migrations as expected. See section 6.2.5 for details.

5.4.3 Version 3: Adding Adaptivity to Model Dynamics

Not only the migration of model and simulation entities costs time, but also the load balancing procedure itself. This cost may hamper the simulation performance, particularly if a large and complex model gets load balanced again and again, without actually having to migrate anything. Therefore, the load balancing algorithm should adapt itself to the dynamism and the complexity of the running simulation.

Before this can be discussed, it has to be decided which measurement is used to determine when the load balancing algorithm should be invoked. The progress of a PDEVS system can be measured by simulation time and wallclock time. However, using either of them causes serious problems. If the simulation time is used to trigger load balancing and it advances erratically, the load balancing may be invoked after too few or too many actual computations. To use the wallclock time instead might be misleading in a similar way. Although it is possible to control the amount of computation between each load balancing invocation by using wallclock time, this is not sufficient for the load balancing algorithm as described in sections 5.4.1 and 5.4.2. The prototype algorithm identifies the inherent parallelism of a model by *counting* the number of parallel executions. When a model is quite complex, the wallclock time interval might be too short to

Algorithm 2 Load Balancing Algorithm

```
double c ← 23 //Communication time constant

int p ← 5 //Number of processors

list setGains ← [] //Set ranking

function executeLoadBalancing
  List gains ← calculateGains()
  sort(gains)
  gains ← subList(gains, 0, p−1) //Remove irrelevant elements
  abstractMapping ← getMapping(gains)
  realMapping ← vote(abstractMapping)
  executeMigration(realMapping)
return

function list ← calculateGains
  list setGains ← []
  foreach coordinator ∈ Coordinators {
    graph ← createParallelismGraph(getLBInfo(coordinator))
    foreach edge ∈ graph
      calculateGains(edge)
    colourSets = colourise(filterNegativeEdges(graph))
    foreach set ∈ colourSets
      setGains ← add(setGains, set, calcGain(set))
  }
return
```

gather *representative* information about parallel executions. In the worst case, load balancing would be executed for *each* simulation step, and the parallelism detection of the algorithm is reduced to absurdity.

This example demonstrates the importance of the simulation step number for a meaningful execution of the load balancing algorithm. Since this number also indicates progress of the simulation (albeit not necessarily progress of simulation time) and the wallclock time, this measurement seems to suit best to the purpose of controlling load balancing invocation.

Obviously, there is a trade-off between calling the load balancing mechanism too often (high costs for load balancing) or too seldom (no effect on speedup). To investigate a simple adaptation scheme, the algorithm re-calculates the number of simulation steps between two invocations as follows:

$$stepCount = \max(1, stepCount) \cdot \min(1 + c, \max(1 - c, \frac{approvedMig_{rate} \cdot n}{\max(1, migrationNum)})) \qquad (5.9)$$

, where n is the overall number of DEVS models (atomic or coupled), $migrationNum$ is the number of models that were moved during the last execution, $approvedMig_{rate}$ is the *desired migration rate*, and c is the *change parameter*. The expression is re-calculated after each execution of the algorithm. The idea is to configure the load balancing execution, so that it migrates a certain amount of models for each step. Since model tree size may vary, this amount is given as a rate $approvedMig_{rate}$, relative to the number of models (n). Setting $approvedMig_{rate} = 0.1$ means that 10% of all models should ideally be migrated per load balancing invocation. Of course, this 'ideal' migration rate may depend strongly on the application, and it should be automatically found by the algorithm. For this prototype, however, it is assumed that there *is* a fixed number of migration operations that would be ideal. This facilitates controlling the adaption capabilities of the algorithm.

The change parameter c is used to control the dynamics of $stepCount$. If $c = 0$, $stepCount$ will not be changed at all, so there is no adaptive behaviour. In contrast, if $c = 0.5$, $stepCount$ may be multiplied by any value in $[0.5, 1.5]$. If $c > 1$, this only has an impact on the min term of equation (5.9), since the fraction will always be positive. However, $stepCount$ could be set to 0 in this case. To ensure that the algorithm is able to 'escape' from this situation, the whole expression is multiplied by $\max(1, stepCount)$ instead of $stepCount$. The variation of c allows to control how the dynamism of the adaptation relates to the dynamism of the adaptation subject. This will be investigated in chapter 6.

The adaptation mechanism presented here has several flaws. Besides its dependence on two additional parameters, it could be observed – by using SIMSIM's interactive simulation mode – that it tends to oscillate. This is caused by the integration of migration cost with the gain calculations.

Consider a very stable simulation run. The adaptation mechanism will increase $stepCount$. At some point in time, the migration cost will be insignificant with respect to the gain, even for models that do not run in parallel very often. This is not a problem by itself, since the algorithm should *indeed* migrate these models after observing them and noticing that their behaviour is stable.

But using the adaptation mechanism as defined before, these migrations are likely to be undone during the next load balancing. This happens because the adaptation mechanism reacts to the *increased* number of migrations by decreasing $stepCount$. Now that $stepCount$ is smaller, the migration costs are not insignificant anymore. Hence, the models in question will migrate *again*.

Therefore, the presented approach does not seem to be a very good solution. Anyhow, SIMSIM was able to detect the design flaw *before* it has been implemented and tested in JAMES II (see figure 6.9 in section 6.2.5). In section 6.3, it is shown that this behaviour does indeed occur in JAMES II as well.

Of course, there are many possibilities to overcome oscillations and other problems (see section 5.5). For example, $stepCount$ could be buffered, so that a single above-average amount of migrations will not affect the $stepCount$ calculation. Since this is a *prototype* implementation, it was left as is.

5.5 Possible Enhancements

The algorithm described in section 5.4 still has much potential for optimisations. First of all, some very important circumstances, like different processing and communication capacities of the environment, are not yet taken into account. So, the algorithm in its current state would only perform well on homogeneous setups. Furthermore, the size of x- and y-messages is not recorded, so that communication requirements regarding bandwidth cannot be determined.

Even *if* these factors are negligible, the gain calculation still only *estimates* the gain. This estimation is based on the assumption that the current behaviour of a model resembles to its future behaviour, which is not necessarily the case. Hence, the load balancing algorithm could be enhanced by pattern prediction mechanisms to generate more sophisticated placements. Moreover, simply summing up the estimated gain of edges between models that could be separated does *not* calculate the *real* gain – it is just an estimation and could be optimised as well. An accurate calculation would require *all* information about model executions, but this is not stored. Instead, parallelism matrices are used to avoid such a memory overhead.

During the gain calculation, it is also implicitly assumed that the execution times of a model's state transition functions are evenly distributed, so that the average execution time used in section 5.4.2 is similar to the average execution time of *parallel* executions. This assumption does, for example, not necessarily hold true if the confluent state transition function of an atomic model needs significantly more or less time than the others. So, the *parallel* execution times should be recorded for *each* parallel execution. Then, they could be used like the values of the parallelism matrices.

Moreover, support for dynamic (P)DEVS models and constraint definitions (as used by the DEVS partitioning algorithm described in [29]) is missing. It is also possible to optimise the execution of the load balancing algorithm. The algorithm could be executed in a single thread running in the background, while the simulation is progressing. When it finishes, the migrations are executed and the simulation would only have to stop for this (relatively) short moment. Finally, the gain calculation shows that the placement of root coordinator and topmost coordinator is *not* irrelevant, but that it should be placed on the processor with the largest computing capacities (see section 5.4.2).

Even though the algorithm is not complete yet, it is already complex and hard to analyse. Since this work focuses on the *simulation* of load balancing algorithms and the possible benefits of doing so, the algorithm will not be developed further in this context. However, the algorithm's simulation might identify the most promising enhancements to be done in future work.

5.6 Porting the Algorithm to James II

The implementation of the load balancing prototype in James II introduced several new problems. One of the most important requirements for a component in James II is that it preserves the flexible architecture of James II's core classes. This is usually done by developing a framework which generalises all mechanisms that follow the same purpose (e.g. partitioning or load balancing). However, since this algorithm is merely a prototype, an easier to develop but more specific and inflexible implementation of the algorithm was integrated with James II.

Nevertheless, some other problems had to be solved. For example, the nodes of the model tree are numbered differently in SimSim and James II, so mappings to convert the data structures had to be generated. This allowed to simply replug the load balancing algorithm from SimSim to James II. However, a basic migration mechanism had to be implemented, and the algorithm to obtain execution times and parallelism matrices needed some adjustments. Until now, no mechanism to automatically measure the communication capacity between two processors has been developed, since this is a rather complex task. Instead, this value is estimated for the current environment and hard-coded within the load balancing implementation. This was also done for migration cost, which may largely depend on the communication capacities of the environment.

6 Results and Analysis

This chapter details some experiments to test the load balancing prototype's performance. At first, a basic experimental setup is worked out (section 6.1). This setup will sometimes be modified to enhance the expressiveness of the following experiments, which are described in section 6.2. The chapter is concluded by section 6.3, which provides a rudimentary comparison between predicted and actual performance of the load balancing algorithm, and a brief analysis of SimSim's performance (section 6.4).

6.1 Experimental Setup

There are diverse parameters that can be adjusted for each experiment. The number of state phases that describe an atomic model's behaviour is theoretically unlimited, so that the number of given parameters can be arbitrarily high (if the available memory is arbitrarily large, too). Even without considering this circumstance, there are at least 20 parameters to be set for each run (see table 6.1). Not all of them can be investigated thoroughly in the context of this work.

Instead of trying to do so, I focused on the algorithm's behaviour under certain circumstances of importance. The requirements for a load balancing algorithm have already been defined in section 5.2. Section 5.4 further narrowed them down: only the load balancing quality (i.e. the model placement) and the algorithm's ability to adapt itself to its environment are observable by SimSim. The measurement and accumulation of run-time information are indeed challenging tasks, but they were assumed to be working accurately (see section 5.4). The fifth requirement, being able to migrate model parts, cannot be evaluated properly by SimSim as well. In SimSim's James II model, each model migration simply takes a certain amount of time. So, migration is not explicitly modelled and can therefore *not* be tested with respect to its efficient functioning.

To facilitate the result analysis, the impact of several parameters is eliminated. This reduces the number of parameters on which the performance results depend. The `DefaultHardwareModel` is used to assign the logical processes to physical processors. It places each LP on a single processor and models a complete graph as network topology. Besides that, all connections are assumed to be of the same quality, so that the two parameters T_s and T_w suffice to describe all communication cost (see section 4.6). Otherwise, they would have to be defined for each network connection. Furthermore, a *homogeneous* environment is assumed. This assumption ensures that the algorithm can be tested in spite of missing support for *inhomogeneous* environments (see section 5.5). So, the computation power factor cp_i is set to `1.0` for each processor i.

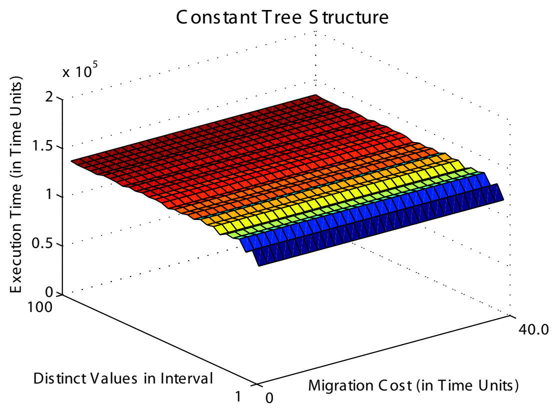
Until now, all time parameters have been described without giving an explicit time unit. Since SimSim's time calculations are valid for *any* time scale, it is possible to use the most suitable time scale for the task at hand. For example, it might be advantageous to configure a simulation of a grid simulation (large latency, etc.) in seconds or milliseconds, while the execution on a symmetric multiprocessor should be parameterised using microsecond precision. As long as all parameter settings adhere to the *same* time scale, the simulation will be valid. To highlight this fact, all concrete time parameters are given in *time units* (tu). Moreover, application model time and simulated wallclock time have to be differentiated (see section 4.1). Since both times are independent and may be scaled *differently*, the time units tu_{amt} and tu_{swt} are introduced. They represent the units of application model time and simulated wallclock time, respectively. Hence, if tu_{amt} means hours and tu_{swt} means microseconds, this would result in simulating a simulation execution rather accurately ($tu_{swt} = \mu s$), while the application model makes use of a more coarse-grained time scale ($tu_{amt} = h$).

Besides that, the execution time of the load balancing algorithm is deemed to be negligible and thus set to

`0.0` tu_{swt}. Nevertheless, the cost of accumulating the information is reflected in traversing the model tree, which may cost a considerable amount of additional time, depending on the network latencies. Moreover, migrating a model element (and its associated simulation component) is assumed to cost `5.0` tu_{swt}, which equals the transfer time of five messages: T_s is set to `1.0` tu_{swt}, while T_w is set to `0.0` tu_{swt} for all network connections, so that the message transfer time is constant. This migration cost value was chosen because at least *three* messages have to be sent in order to migrate a PDEVS model: One from root coordinator to host LP, to trigger the migration. Another message is needed for the actual migration, and in the end, the root coordinator has to be notified about completion, so that the simulation can proceed. Choosing `5.0` tu_{swt} instead of `3.0` tu_{swt} accounts for the possibility of additional messages caused by more complicated migration protocols. For example, the host LP may have to request a migration acknowledgement from the destination LP before it sends the actual data, or references have to be updated. All in all, only the *pure* execution time of the load balancing algorithm is neglected.

The benchmark model parameters are simplified as well. By default, only one state phase is defined. It determines the overall behaviour of *all* atomic models. Consequently, the number of rounds to be executed is set to `1` and P_{change}, the probability that a model changes its group after finishing the current state phase, can be set to `0.0`. Since T_w is set to `0.0` tu_{swt}, the message size $\Delta_{msgSize}$ is negligible.

The model tree consists of `100` nodes and has a branch factor of `4`. To avoid additional noise generated by a stochastic tree generation, the same random tree is used for all experiments, if not stated otherwise. The results are still valid, because the number of nodes and the branch factor are the decisive factors regarding the model tree, and not its actual structure. The extent of noise introduced by randomising the actual tree structure is depicted in figure 6.1.



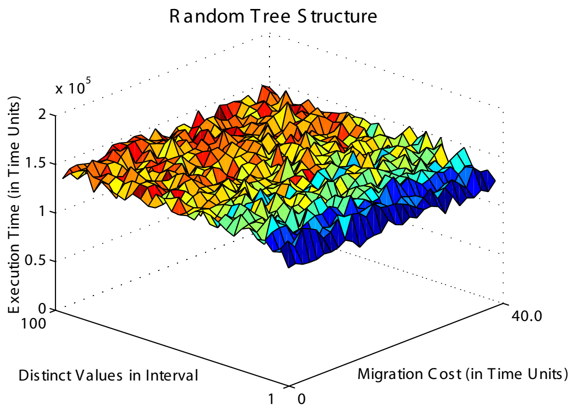


Figure 6.1: Noise introduced by Random Tree Structure. Both curves show the same properties, but the data obtained by using a constant tree structure is easier to analyse. These results were obtained by running the same experiment as described in section 6.2.5, but using a longer benchmark model run ($timeSpan = 60$, $rounds = 2$) and another message interval distribution ($U_r(1.0, 2.0, X)$, X being one of the varied parameters).

Finally, the default settings define a rather optimistic scenario for parallel execution: Using a rounding uniform distribution of $U_r(0.1, 1.0, 1)$ means that all message intervals are either `0.1`, `0.55`, or `1`. This results in a high degree of inherent parallelism. Setting $pref$ to `1.0` causes all models to communicate exclusively with other models in their group, so that the communication between atomic models is strongly clustered. Moreover, the execution of each internal and external state transition function is assumed to take `10.0` tu_{swt}, so that the granularity for internal and external state transition functions is `10.0` (see section 2.2.2, recall that $T_w = 0.0$ and $T_s = 1.0$). Since the confluent state transition function subsequently executes an internal and an external state transition, its granularity is even `20.0`. However, *four* messages are involved in each state transition. Hence, the *real* granularity is much lower: `2.5` for internal and external state transitions, and 5.0 for confluent state transitions. Considering that a granularity of 1.0 or less means that *no* parallel speedup is possible in principle, these values are not too optimistic. The parameters and their default values are summarised in table 6.1.

6.2 Simulated Performance Results

6.2.1 Granularity

As already described in section 2.2.2, granularity is essential for the efficiency of distributed simulation. Therefore, it is particularly interesting to investigate how the load balancing algorithm performs under different granularity conditions. To measure this, both $\Delta_{\delta_{int}T}$ *and* $\Delta_{\delta_{ext}T}$ were varied from $U(0.0, 0.0)$ tu_{swt} to $U(100.0, 100.0)$ tu_{swt}. Additionally, T_s was varied from 0.0 tu_{swt} to 30.0 tu_{swt}.

Figure 6.2 shows the execution times of three different setups: A sequential simulation without any load balancing, a distributed simulation using the basic version of algorithm (i.e. a version without migration cost consideration, voting, and adaptation), and another distributed simulation using an 'enhanced' version of the algorithm. The enhanced version considers migration cost and uses the voting scheme, but does not adapt its load balancing frequency, since this feature might extremely influence the algorithm's behaviour (see figure 5.5). As a matter of course, the sequential execution does not depend on the message transfer time. The predicted sequential execution time grows linearly when the required computing effort is increased.

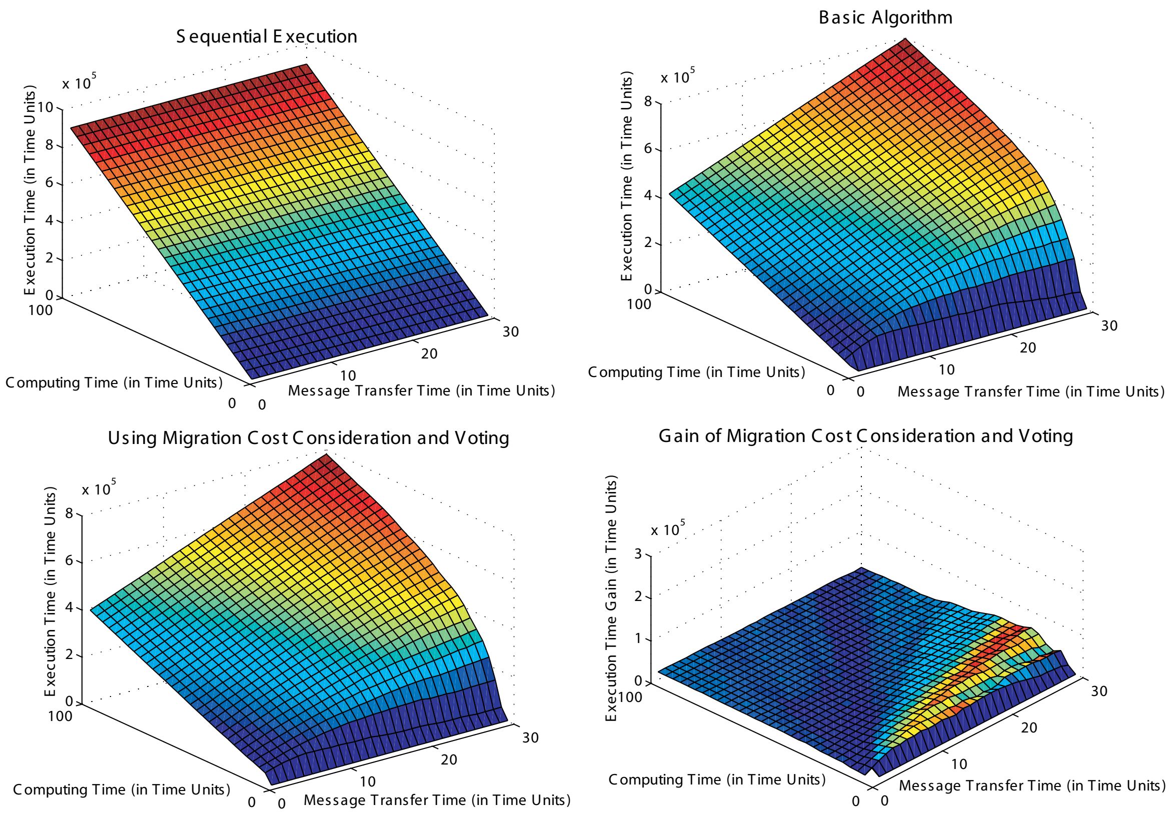


Figure 6.2: Effect of Granularity on Execution Time

As could be expected, the performance of the basic load balancing algorithm is good when computing

Name	Value	Description
SimSim		
cp_i	`1.0`	Each processor i has the same computation power factor. Furthermore, each LP runs on a single processor.
T_s	`1.0` tu_{swt}	Message start-up time, given in time units of simulated wall-clock time. This value is used to determine the message transfer time between *distinct* processors. For intra-processor messages, T_s is assumed to be `0.0`.
T_w	`0.0` tu_{swt}	Additional message transfer time per byte.
Model		
$numOfGroups$	4	Groups in the benchmark model.
$rounds$	1	Number of rounds for each model's state phase execution.
$numOfNodes$	100	Number of models in the DEVS model tree.
$branchFactor$	4	Branch factor of the DEVS model tree.
$numOfLPs$	8	Number of available LPs.
$lbCost$	`0.0` tu_{swt}	Execution time of load balancing algorithm.
$lbFrequency$	10	The load balancing algorithm will be invoked after this number of *-messages has been processed. If the adaptation mechanism is used, this is the *initial* load balancing frequency.
$migrationCost$	`5.0` tu_{swt}	Additional time needed to migrate one node of the model tree.
$partitioning$	DEVS partitioning	The DEVS partitioning algorithm from [29] is used to create an initial partition of the model tree.
State phase		
$timeSpan$	`30.0` tu_{amt}	Time span of the state phase, given in time units of application model time. All atomic models are parameterised using this single state phase.
$\Delta_{nextMsgT}$	$U_r(0.1, 1.0, 1)\ tu_{amt}$	Time between sending new messages, given as a rounding uniform distribution between `0.1` and `1.0`, using `1` intermediate value, which is `0.55`.
$\Delta_{reactionT}$	$U(1.0, 1.0)\ tu_{amt}$	Reaction time that passes until a message is sent back to its origin.
$\Delta_{msgSize}$	$U(100, 100)$ byte	Size of a new message, given as a uniform distribution that returns a fixed value of `100`. It is irrelevant in this context, since T_w is set to `0.0`.
$\Delta_{\delta_{int}T}$	$U(10.0, 10.0)\ tu_{swt}$	Computing time for an internal state transition.
$\Delta_{\delta_{ext}T}$	$U(10.0, 10.0)\ tu_{swt}$	Computing time for an external state transition. For confluent state transitions, the execution time is the sum of internal and external state transition time.
P_{change}	`0.0`	Probability that a model changes its group after finishing a state phase. Since there is only one state phase and $rounds = 1$, this parameter is irrelevant in this context.
$pref$	`1.0`	Preference of sending new messages to models in the same group. A value of `1.0` means that all models *exclusively* communicate with models in their group.

Table 6.1: Basic Experiment Parameters. This table lists all parameters and their values for the experiments described here. In the following, it will only be described in which way parameter configurations *differ* from these settings. Detailed information on the parameters is given in section 4.4 and 5.3.

requirements are high and network latencies are low. If the computing time is small and the message transfer times are sufficiently high, the gain calculations as described in section 5.4.2 may result in negative values. If all gains are negative, all models are moved to the root coordinator's LP. This explains the basic algorithm's performance for setups with an extremely low granularity, i.e. with a large message transfer time and a low computing time. In this region, the execution time is *not* proportional to the message transfer or computing time anymore.

The performance of the enhanced version is very similar to the performance of the basic algorithm. To scrutinise the gain[1] of considering migration cost and using the voting scheme, the enhanced algorithm's execution times were subtracted from the ones of the basic algorithm. The result is depicted in the bottom right corner of figure 6.2. It can be seen that using the enhanced algorithm *is* advantageous, because all values are greater than `0`. On the other hand, the gain is rather small in most setups. Only when the computing times are sufficiently small, voting and migration cost consideration seem to pay off.

In section 3.4, I argued that each way of simulating an application model may be optimal in certain *regions* of the simulation space. Although the simulation space is multidimensional, SIMSIM allows to identify the borders of these regions. Since each experiment only varies two parameters along two dimensions (e.g. in this case, computing time and communication speed), the observed borders will be 2-dimensional as well. In fact, they can be regarded as 2-dimensional 'cross-over points' (see section 3.4 and figure 3.3). Figure 6.3 depicts the *gain regions* of a distributed PDEVS execution using the load balancing algorithm, in comparison to a sequential execution. The gain was again calculated by simply subtracting the execution times of distributed execution from the execution times of sequential execution. So, all positive values represent a simulation run which could be completed faster by using distributed execution, and the negative ones stand for situations in which a sequential execution was better.

The upper left plot shows the gain of the basic algorithm, which is very similar to the gain of the enhanced algorithm (not depicted). In addition to the experiments defined at the beginning of this section, the enhanced version of the algorithm was also tested by using an application model with less inherent parallelism, i.e. $\Delta_{nextMsgT}$ was set to $U_r(0.1, 1.0, 5)$ instead of $U_r(0.1, 1.0, 1)$. The upper right plot in figure 6.3 shows how this parameter adjustment 'affects' the border between profitable and unprofitable distributed execution. Actually, the parameter adjustment does not *affect* the border of the region. Instead, a *different* part of the simulation space is investigated, in which the border of the gain region is positioned differently. The plot also indicates that the enhanced algorithm, even though it considers migration costs and uses the voting scheme to minimise migrations, is still too optimistic. Otherwise, it would not be so unprofitable under certain conditions. A load balancing algorithm that correctly adapts to the granularity of the simulation system would place all model entities on one host as soon as the granularity is too small to yield speedup. Hence, using such a load balancing algorithm in low-granularity scenarios would not be such a problem. The gain of the developed algorithm, however, decreases steeply under these conditions. These results motivate a reconsideration of the gain estimation method, which may improve the algorithm's performance in the future. Interestingly, the advantage of the enhanced algorithm over the basic one (see gain curve in figure 6.2) is only relevant in a region where *no* distributed simulation is able to get any speedup, because the granularity is less than `1.0`. So, it can be concluded that the extensions of the algorithm do not significantly *contribute* to speedup in this context, but rather cut the losses in regions of the simulation space where distributed simulation is unfavourable.

Besides that, it has to be said that these experiments are *still* rather optimistic, not only because of the high parallelism, but also because the parameter $migrationCost$ is assumed to be `5.0` tu_{swt} per default. For the tested setups with a message transfer time of up to `30.0` tu_{swt}, this value is unrealistically small[2].

Furthermore, the performance of the enhanced algorithm using adaptation with $c = 0.1$ was tested (using the old parameters again). As can be seen at the bottom of figure 6.3, the introduction of adaptation is *very* disadvantageous for the tested setups. The gain curve of this version of the algorithm *completely* differs from the others. Most stunningly, it performs particularly bad when the computing time increases, which actually should be *advantageous* for distributed simulation. This plot indicates that c might have

[1]This gain must not be confused with the notion of gain employed for the load balancing considerations.

[2]Being unrealistic does not mean that the experiments are invalid. These parameters suit well for models that exchange large messages and manage small states, but this type of model might just be not very representative.

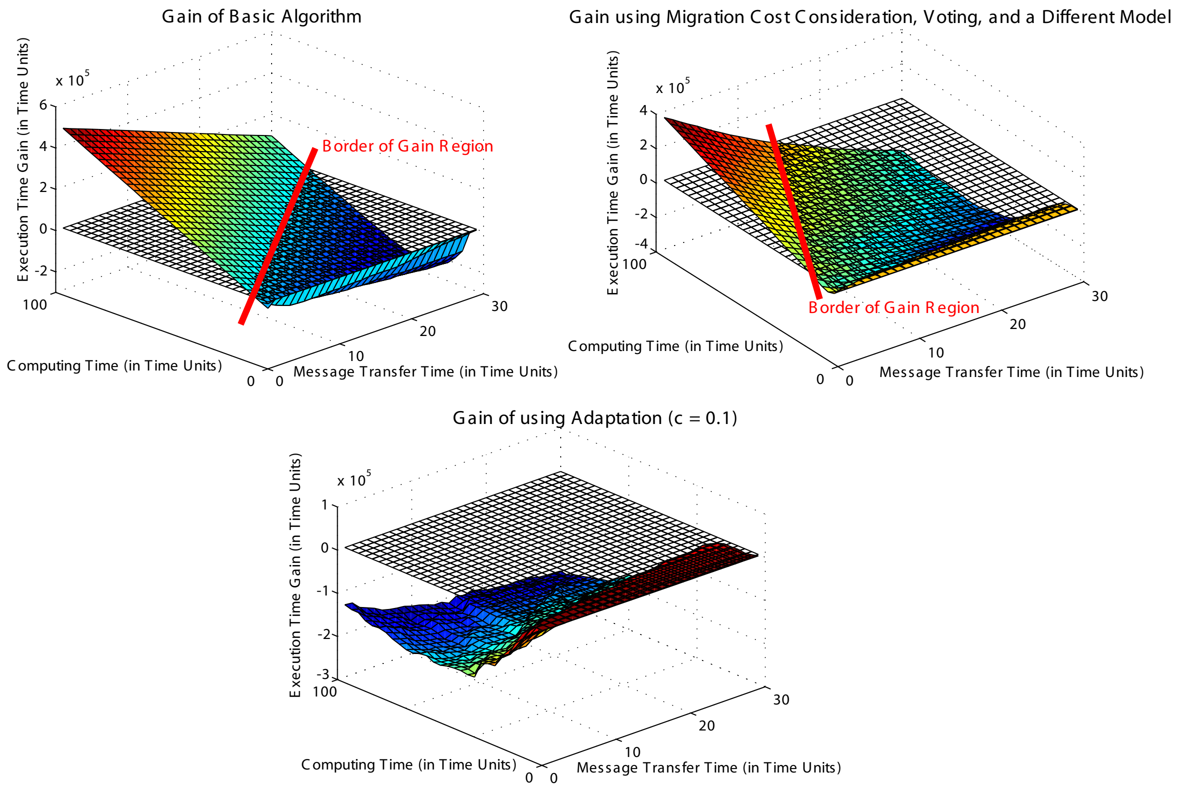


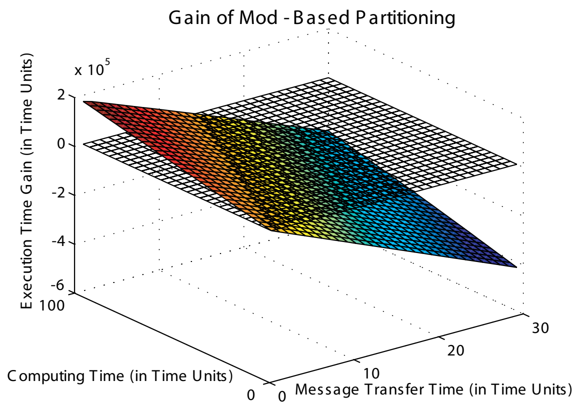
Figure 6.3: Gain Regions of the Load Balancing Algorithm. The wireframe represents the border between faster sequential execution (negative values) and faster distributed execution (positive values).

been set too high, and thus the load balancing algorithm with its time-consuming information fetching phase is called too often under certain circumstances.

Another interesting question to be asked concerns the actual gain of using the load balancing algorithm when executing a distributed simulation. Figure 6.4 shows the gain when running the simulation without any load balancing mechanism, but still in a distributed manner. On the left side, the gain of an distributed simulation using a modulo-based partitioning scheme is depicted. This means, the position of each model is determined using its ID (as described in section 5.3) and calculating the modulo function with respect to the number of available LPs. The result is the index of the LP to which the model is assigned (presuming that LP indices start from 0). Evidently, the execution time gain is only positive for high computation costs and small communication costs. On the right side, the performance when using a partitioning algorithm tailored to DEVS models is shown[3]. As can be seen, the gain region of the partitioning algorithm *without* any load balancing algorithm is *larger* than the gain region when used in combination with the load balancing algorithm.

While this might look like a devastating result at first sight, it has to be stressed that the experimental setup is *optimal* for initial partitioning and does not require the particular advantages of load balancing: All models induce the *same* computing load, and this load is *constant* during a simulation run. As soon as the required computation time *changes* dynamically, initial partitioning will not suffice. By using different state phases, SimSim could be used to find out how much dynamic is needed until the load balancing mechanism is beneficial. However, since the number of parameters to be investigated is already quite large, even without using multiple state phases, this work has to be continued in another context.

[3]The partitioning algorithm is rather complex, so further details have been left out here. The algorithm is detailed in [29].



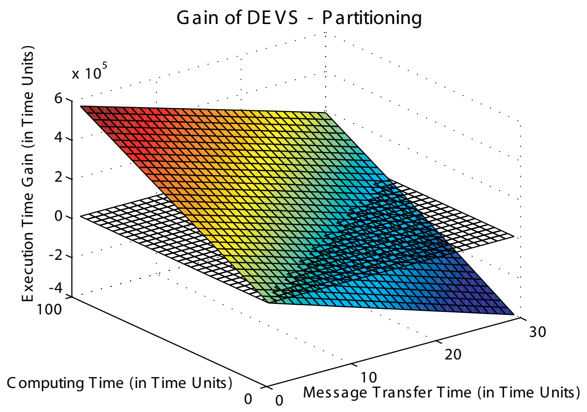


Figure 6.4: Comparison to Distributed Simulation Without Load Balancing

6.2.2 Parallelism and Migration Cost

To test the adaptivity of the algorithm with respect to other parameters, the migration cost was varied from `0.0` tu_{swt} to `40.0` tu_{swt}. Besides that, the distribution of time intervals between newly generated text messages was varied between $U_r(0.1, 1.0, 1)$ tu_{amt} and $U_r(0.1, 1.0, 100)$ tu_{amt}. Only the amount of distinct values in the distribution's value interval was adjusted. As already explained, this serves as a mechanism to control the inherent parallelism of the model. The migrations caused by different versions of the load balancing algorithm during these experiments were already depicted in figure 5.5. Figure 6.5 shows the corresponding execution times.

As could be expected, the sequential simulation runs were not affected by any of the parameters, neither by the inherent parallelism, which they are not able to exploit, nor by the migration costs, which do not occur. There is a *slight* slope from models with a few values in the U_r interval (i.e. well parallelisable) to models with many values in this interval (i.e. not so well parallelisable). This slope is probably an artefact induced by the way the rounding distribution determines the values. In the upper right corner, the gain of using a distributed execution with the basic load balancing algorithm, instead of a sequential execution, is presented. Since migration costs are not considered, the algorithm performs particularly bad when migration costs are high. This behaviour can be avoided by using the enhanced version of the algorithm, which adapts rather successfully to these parameters (bottom left corner). Even the algorithm using adaptation performs quite well and its loss with respect to sequential execution is smaller than the loss of the other variants (by nearly one order of magnitude). However, considering the results of section 6.2.1, the distribution and amount of caused migrations (see figure 5.5), and its 'abnormal' behaviour when the computing time is increased (it causes the algorithm's performance gain to *drop*!), these results should not be generalized and may only be valid for certain low-granularity regions of the simulation space.

6.2.3 Scalability

Another interesting requirement whose fulfilment can be analysed beforehand is the scalability of a distributed PDEVS simulation that makes use of these algorithms. In section 3.1, Nicol's very sophisticated analytical approach to investigate scalability was introduced. Using SimSim is another alternative of getting an idea about the scalability of a PDES system, including the parameters that influence it. Although these results are not as comprehensive as the results of a mathematical analysis would be, they could be generated fast and without any additional efforts.

Since the enhanced version of the algorithm (that is, the algorithm using migration cost consideration and voting, but no adaptation) performed best so far, it was used for these experiments. As could be seen in section 6.2.2, the default parameters do not introduce sufficient granularity to allow *any* speedup.

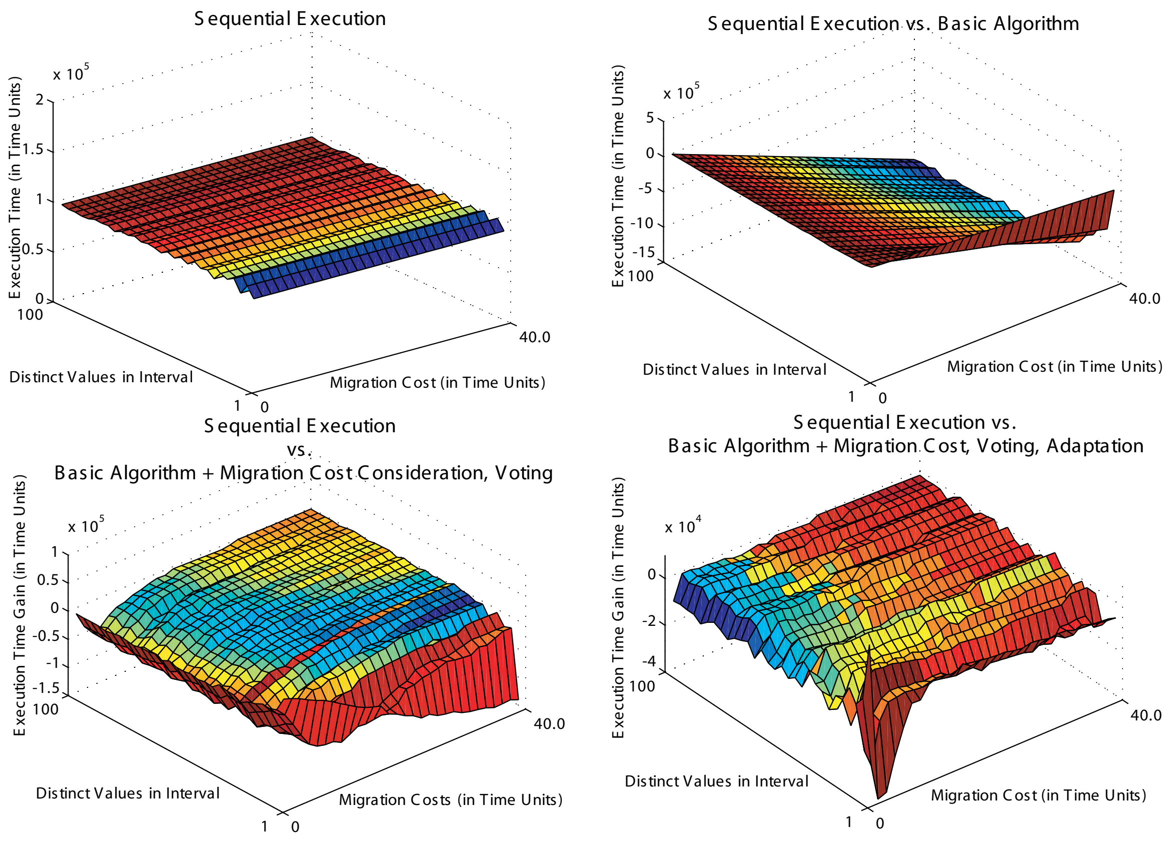
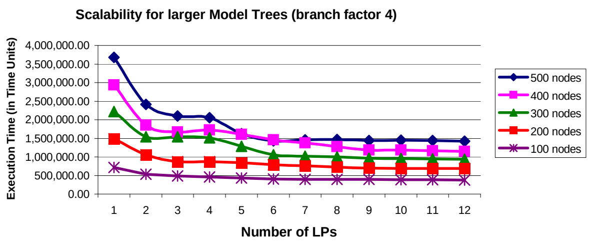


Figure 6.5: The Effect of Parallelism and Migration Cost

Therefore, the computing time of the internal and external state transition functions, $\Delta_{\delta_{int}T}$ and $\Delta_{\delta_{ext}T}$, are both set to $U(20.0, 20.0)\ tu_{swt}$. Moreover, the application model runs twice as long, i.e. $rounds$ is set to two instead of one.

Figure 6.6 shows the scalability with respect to different sizes of the model tree, ranging from 100 nodes to 500 nodes. Two different branch factors were used, four and eight. The branch factor determines the average number of children per coupled model in the model tree, and thus the 'width' of the tree. All in all, the execution time was predicted for eight different model trees, running on one to twelve LPs. The plots suggest that a higher branch factor increases the scalability of distributed PDEVS simulation. In the left plot, depicting the runs with trees having a branch factor of four, nearly no speedup is gained when running on more than six LPs. In the right plot however, the execution of large trees also experiences a speedup, albeit quite moderate, when varying the number of LPs from six to twelve. So, not only the tree size affects the scalability of the simulation, but also the branch factor. This dependency might be caused by the basic load balancing approach of analysing parallelism on a single level: The higher the branch factor, the larger are the parallelism matrices. This might yield a better model placement.

Figure 6.7 shows the scalability of the default model tree regarding different degrees of parallelism. Evidently, inherent parallelism is a major prerequisite of a scalable, distributed PDEVS simulation. However, this dependency may not be as strong for larger model trees. In larger trees containing more atomic models, e.g. six distinct values per interval might suffice to cause enough messages to be sent in parallel, so that a distributed simulation is more advantageous. The plots also support the conclusion that the enhanced version of the algorithm adapts to parallelism rather well, since a low degree of parallelism does not cause the execution times to rise, but to stay constant instead. This illustrates that the algorithm does indeed



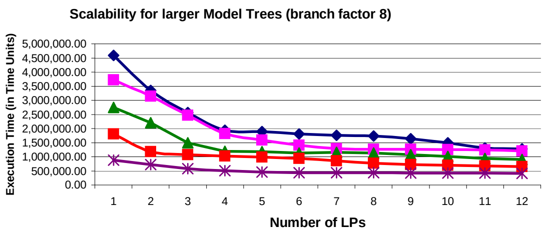


Figure 6.6: Scalability and Tree Size.

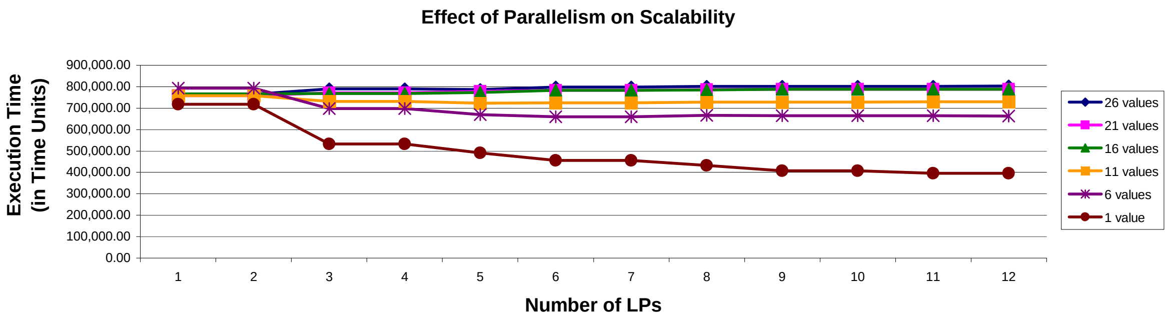


Figure 6.7: Scalability and Parallelism

resign a distributed simulation and accumulates all models on one host if everything else does not make sense.

6.2.4 Dynamism

The effect of dynamism was tested by reducing the time span of the default state phase to 3.0 tu_{awt} and increasing the number of rounds to 20. As in section 6.2.3, the computing time for the internal state transition functions was set to $U(20.0, 20.0)$ tu_{swt}. Then, the change probability was varied from 0.0 (no atomic models change their group) to 1.0 (all atomic models change their group 20 times). The preference factor $pref$ was varied from 0.0 (messages are sent to arbitrary atomic models) to 1.0 (messages are only sent to models in group) as well. This experiment was done to test the enhanced version of the algorithm, and the version that additionally uses the adaptation mechanism.

As can be seen in figure 6.8, the varied parameters do not influence the execution times of the algorithms at all. Such a result is also very useful, since it rules out two parameters that might have to be considered more thoroughly otherwise. However, this still is a *diminutive* part of the simulation space to be explored, so that there are probably situations in which these parameters *do* play a role. For example, when a low load balancing frequency is used, the change and preference factors greatly influence the expressiveness of the parallelism matrices.

Finally, it has to be noticed that the algorithm using the adaptation mechanism needs more than twice as much time to execute the simulations than the enhanced algorithm. These results are yet another motivation to revise the adaptation mechanism. The next section gives an interesting hint for corrections.

6.2.5 Analysing the Adaptation Mechanism

As already mentioned in section 5.4.3, the adaptation mechanism tends to cause oscillations when used in a situation where granularity is low. This can be observed closely, as well as the influence of the change

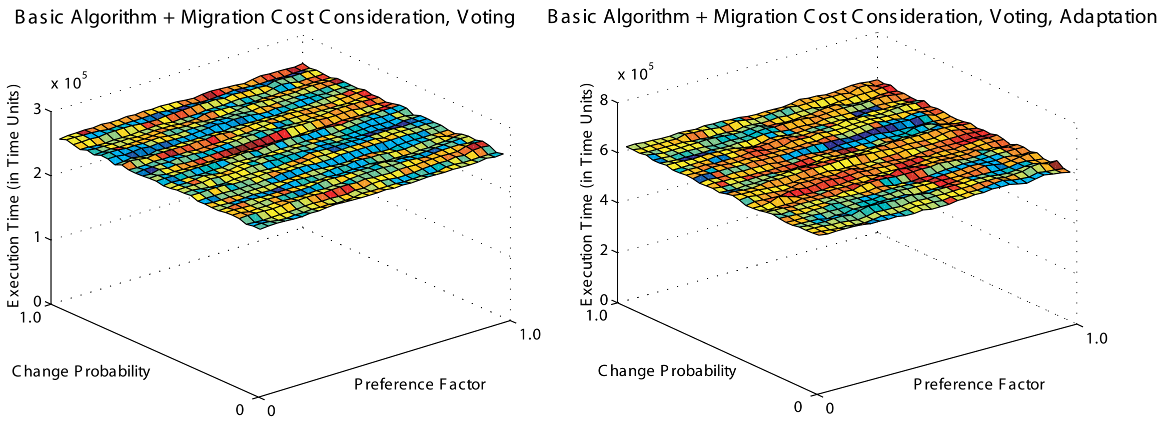


Figure 6.8: Effect of Dynamism on Execution Time

factor c that regulates the adaptation speed. To simulate a low-granularity setup, the computing times of the state transition functions were decreased to $U(1, 1)$ tu_{swt}, whereas the communication cost was dramatically increased by setting T_s to `50.0` tu_{swt}. Since interactive simulation runs had to be done, so that the behaviour of the algorithm during a single simulation run could be scrutinised, SimSim's execution speed was increased by decreasing the size of the model tree to 20. At the same time, the number of $rounds$ was set to 15, so that the simulated wallclock time needed to execute the benchmark model was still long enough to make some observations. The adaptation mechanism's change parameter c was set to `1.0`, `0.5`, and `0.1` respectively.

Figure 6.9 shows the points in simulated wallclock time at which the load balancing algorithm was evoked, and the number of migrations that were caused by it. In the first plot, where $c = 1$, load balancing was invoked quite often. At first, all models are moved to the root coordinator's LP, which is correct, since the granularity is too low to gain speedup. This causes 15 migrations, and since this is much more than 10% of the model tree (which was assumed to be a reasonable value to test the approach, see section 5.4.3), the number of *-messages until load balancing invocation is drastically reduced. Since thereafter no migration has to take place, this number of *-messages will be increased again and again, which can be noted by considering the increasingly large intervals between the load balancing marks. Suddenly, 10 models are moved onto another LP. This happens due to the decreasing importance of migration cost for the gain calculation, as has already been explained in section 5.4.3. Now, the time until next load balancing repetition is quite large. Although the number of *-messages until load balancing invocation has been reduced, which is caused by the high number of migrations (50% of the nodes in the model tree), at least one *-message will be executed in a distributed manner. Since the message transfer time is quite large, this takes much longer than a sequential execution, as can be seen in the plot. Afterwards, all models are re-moved from the remote hosts and placed on the root coordinator's LP again. This oscillation continues.

For the second run, using $c = 0.5$, the situation is very similar. However, the reduced adaptation speed causes a decrease of load balancing invocations and oscillations. While this should be something positive, it turns out that this setup needs much *more* time until the simulation is finished. Since the adaptation speed was reduced, the migration of 10 models does not decrease the number of *-messages until next load balancing as much as in the first run. This causes more *-messages to be executed under very disadvantageous circumstances, which explains the large time gaps between moving and re-moving the PDEVS models during simulation. This is a good example of how counterintuitive the execution of even such a simple mechanism can be.

Finally, if c is set to a value that is small enough, like `0.1`, the oscillations might even vanish. Again, this dos not necessarily mean anything positive. Being able to adjust the number of *-messages by maximal

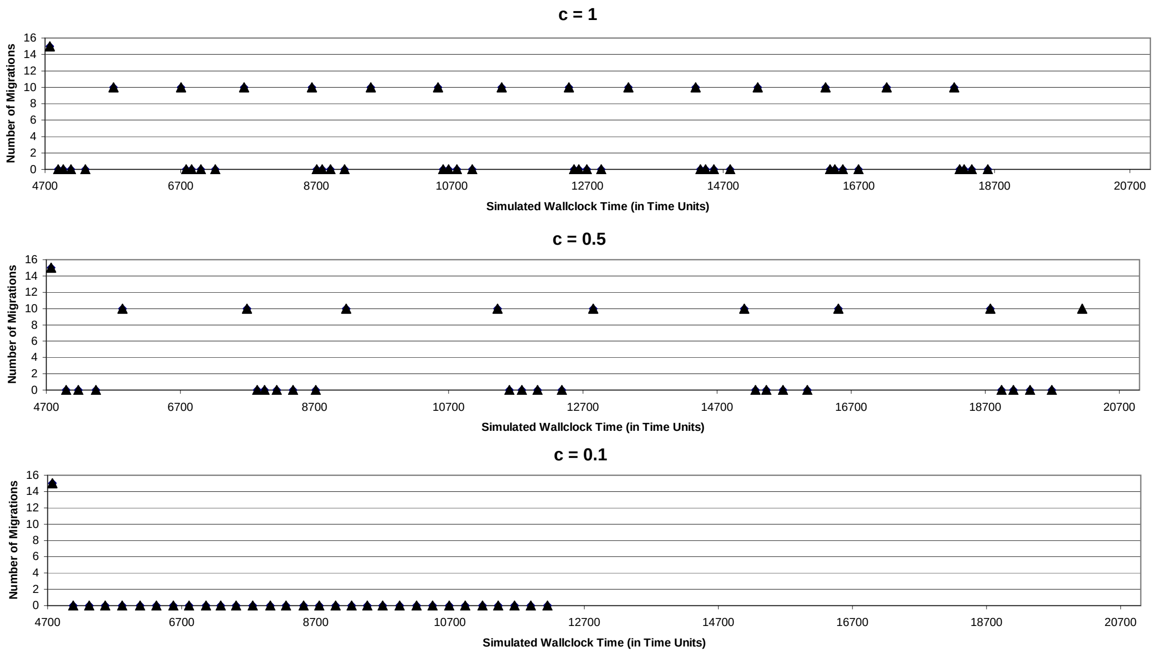


Figure 6.9: Traces to illustrate Adaptivity Mechanism. Each triangle denotes a load balancing invocation, and the number of resulting migrations is shown on the y-axis.

10%, the adaptation mechanism gets 'trapped' at a certain value and is generally unable to adapt to the simulation thereafter. So, this trace actually presents the use of a *static* load balancing frequency. These problems show that the chosen approach has serious drawbacks and can be regarded as not being purposeful. Possible corrections could lead to an algorithm that analyses its previous behaviour and its *success* in the past before it makes a decision concerning a new load balancing frequency.

6.3 Comparison to Performance in James II

Due to lack of time and equipment, only a small-scale comparison between the performance of JAMES II and SIMSIM's predictions could be conducted. Nevertheless, the experiments were set up as carefully as possible, so that some further insights with respect to strengths and weaknesses of this approach can be gained.

The experimental setup consisted of two computers, both running the same *Java Virtual Machines (JVM)* (Sun Microsystem's Java Runtime Environment, version 1.5.0 Update 6 [48]) on Windows XP. One computer had an AMD Athlon64 X2 Dual Core 3800+ processor with a 2 GHz clock rate and 1 GB RAM. This setup achieved a Java SciMark 2.0 score of `224.6` Mflops [2]. The other system was driven by a 1 GHz Intel Pentium III CPU and 256 MB RAM. It achieved a Java SciMark 2.0 score of merely `111.4` Mflops. This is not too problematic for the assumption of using a *homogeneous* system, because the load generation of the benchmark model generates load for a defined *time span*. Thus, both systems execute a transition function for the *same* time, the only difference is that the load generator generates more synthetic load for the faster CPU. However, the Athlon64 system will still execute the rest of the simulation code faster, so that there is a slight performance difference[4].

The systems were connected by a 100 MBit/s Ethernet link, via an intermediate network switch. At first,

[4]Moreover, the processes executed by the Athlon64 CPU had to be configured to run only on *one* of the available cores. Otherwise, the load generation – which is based on CPU time measurements – does not work properly.

the communication cost and the migration cost for this environment had to be estimated. To determine the average communication cost, the benchmark model was executed on the Athlon64 system, using two `SimulationServer` processes (i.e. two LPs). Then, the same experiment was executed with each LP placed one a distinct processor. The execution time difference was divided by the number of exchanged messages. This resulted in a message transfer time estimation of approximately 46.6 ms, which includes all overheads induced by network interfaces etc.. To migrate a coordinator and update all references (see figure 2.3), at least five messages have to be sent[5]. Considering additional efforts to serialise model and simulator components, a migration was assumed to cost eight times the average message transfer time, which totals approximately 373 ms.

In section 6.2, the experiments predict the abstract PDEVS simulator's performance for up to 10^6 time units. To allow an evaluation in a feasible amount of time, the default experiment as outlined in section 6.1 had to be adjusted. The model tree size ($numOfNodes$, see table 6.1) was set to 40, and $timeSpan$ was set to 3 tu_{amt}.

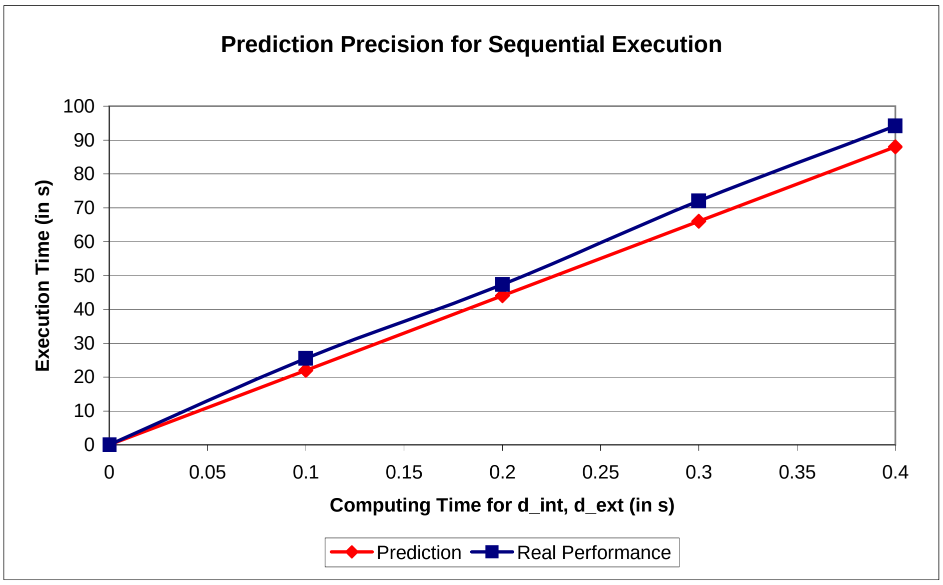


Figure 6.10: Prediction of Sequential Execution Time

Now, SimSim's ability to predict the performance of the abstract PDEVS simulator can be tested. Figure 6.10 compares predicted and real James II performance when executing the benchmark model on a single LP (without any load balancing). As can be seen, the predicted performance is rather accurate. Such an analysis may also help to evaluate different PDEVS simulators regarding possible optimisations. Evidently, the abstract PDEVS simulator implementation in James II works quite efficiently under the observed circumstances.

Some results regarding the parallel execution on two LPs, using the enhanced version of the load balancing algorithm (i.e. migration cost consideration, voting scheme, no adaptation), are depicted in figure 6.11. The computing time of atomic benchmark models, i.e. $\Delta_{\delta_{int}T}$ and $\Delta_{\delta_{ext}T}$, was varied from 0.0 to 0.2 s. Although SimSim's predictions seem quite wrong in these cases, it turned out that these differences were caused by a bug in the load balancing implementation of James II. Indeed, SimSim's predictions could be used to *control* the functioning of the load balancing algorithm's concrete implementation.

The comparison between James II (using a *working* load balancing algorithm) and SimSim's predictions

[5]One to trigger the host LP, one to migrate model and simulator. Then, one message is needed to notify the parent. At least one message is needed to notify the coordinator's sub-models. Finally, the system has to be notified, so that it can proceed.

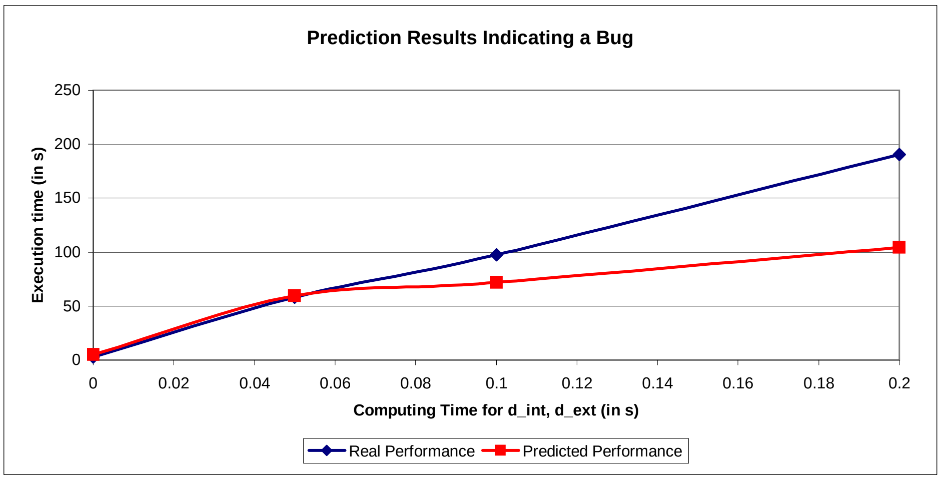


Figure 6.11: Prediction Results Indicating a Bug. A high discrepancy between predicted and actual performance helped to identify a bug on implementation level.

is shown in figure 6.12. The plot on the right shows the difference between predicted and actual performance in relation to the maximum of both values. Although most predictions are acceptable, i.e. they deviate from the actual performance by 10% at most, it can be seen that this is not the case for large or small computing time values. In low granularity scenarios (i.e. when the computing time is small), this is easy to explain: the simulation overhead, which was abstracted away by SimSim's model of James II, is dominating the execution time.

The growing prediction error when *increasing* the granularity is rather counterintuitive. Increasing granularity should lead to a better precision, since the impact of the aforementioned simulation overhead decreases. Recall that all experiments presented in this chapter are initialised with the *same* parameters, except for the variables that are varied to observe their impact on the results. Hence, the execution of the experiments was always the same, except for the synthetic load generation. This implies that overhead induced by James II code – which might be relevant but was not modelled – can only result in a *constant* amount of additional time. However, the difference between real and predicted performance grows *non-linearly* for computing times greater than 0.25 s (see plot on the right of figure 6.12).

The only component whose impact *could* increasingly affect predicted and real performance is the load generation. For experiments regarding the *parallel* James II execution, each coordinator and simulator component was executed in a single thread. While the induced computing times of each thread are considered by SimSim, the time to *switch* threads was not taken into account. The *longer* threads are executed concurrently, the more often the operating system might switch between them. Indeed, this seems to be the factor that hampers the predictability of scenarios with particularly high computing times. To overcome this problem, one could enhance the generic system model by more sophisticated scheduling models (see section 4.4). Simply adjusting other parameters – like the migration cost – will *not* suffice to correct this lack of accuracy, as is illustrated by an additional prediction that assumes a migration cost of 500 ms (see figure 6.12). Having said that, it has to be stressed that this effect will only occur when a large number of threads is used (40 threads on two LPs, in this case), and it 'just' slows down the actual execution, so that its impact is quite foreseeable. Perumalla et al. experienced similar problems caused by operation system peculiarities, albeit on a much smaller scale (see section 3.3, [78]).

Finally, it had to be tested whether the adaptation mechanism's oscillations, which were identified using SimSim, do *really* occur when executing the algorithm in James II. An experimental run was executed, with a benchmark model time span of 100 tu_{amt} and computing time set to 0.02 s. The plots presenting real and predicted migration behaviour are depicted in 6.13.

As can be seen, the oscillations *do* occur as was predicted by SimSim. Nevertheless, the prediction

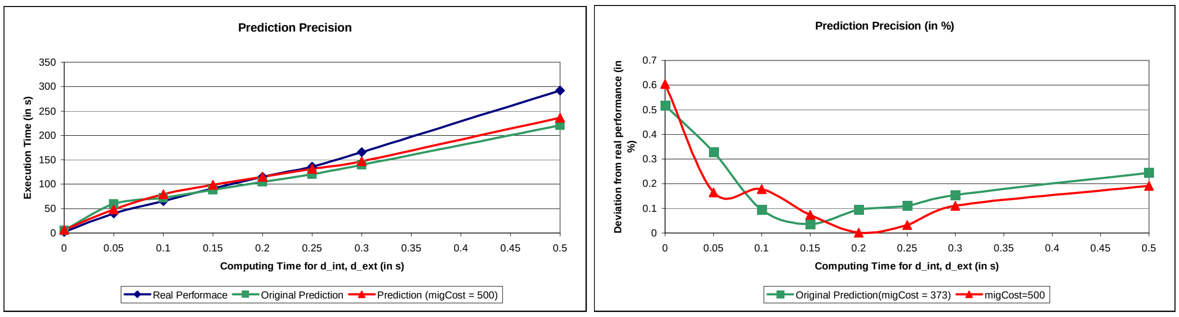


Figure 6.12: Prediction of Parallel Execution Time

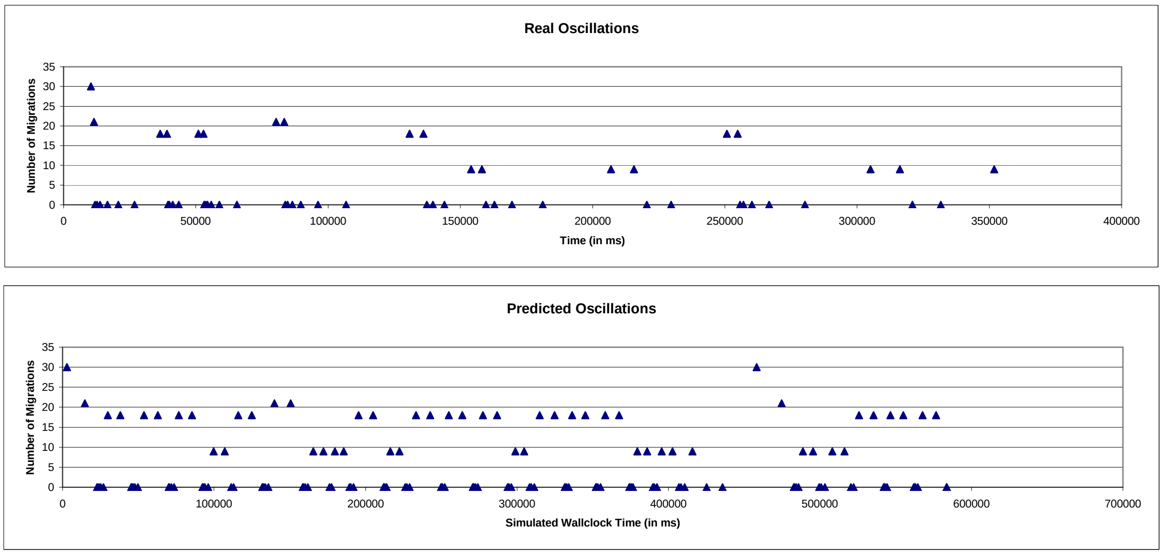


Figure 6.13: Prediction of Unforeseen Behaviour

performance of SimSim is rather bad in this context (note that execution time is predicted to be *twice* as large as it actually is). This could be caused by the very small amount of generated load, since the load generation is not very precise when it comes to generation of fine-grained load.

All in all, it has to be concluded that SimSim's predictions may be false positives or false negatives. However, the factors that cause this behaviour – the impreciseness of the load generator and the impact of thread switching – could be measured and may be integrated with the model in the future. Alternatively, they could be alleviated by using a better load generation mechanism and less threads, respectively.

6.4 SimSim's Performance

To test SimSim's undamped performance, the default experiment has been executed twice. To decrease the influence of object generation overhead, the experiments were configured to run rather long. This was done by setting the number of rounds to 300. Each simulation run consisted of 14,340,950 events.

Altogether, SimSim needed `196.671875` seconds for their execution[6]. Hence, SimSim was able to process approximately `145.000` events per second. The test was executed using Sun Microsystem's JVM for Windows XP, version 1.5.0 (Update 6), on the Athlon64 system that was already described in section 6.3. It has to be noticed that the execution time of the plug-in-specific code is included in these calculations, so that the theoretically possible event processing rate of SimSim's *core* classes might be higher. However, this theoretical value does not have much practical relevance.

[6]This duration was obtained by the class `ThreadMXBean`, which rather accurately determines the processing time of a single Java thread.

7 Conclusions

The hardest thing is to go to sleep at night,
when there are so many urgent things needing to be done.
A huge gap exists between what we know is possible with today's machines
and what we have so far been able to finish.
Donald E. Knuth

This chapter draws some general conclusions and describes the outcomes of this work. The basic facts are summarised in section 7.1. Sections 7.2 and 7.3 put the results into a broader perspective and describe possible subjects of future research.

7.1 Resume

In this work, a simulation-aided method to develop PDES algorithms was motivated (chapter 2 and 3), outlined (chapter 4), and finally put to test (chapter 5 and 6). At first, a variety of PDES systems and the problems they have to deal with were described briefly, so that two problems of developing PDES systems could be illustrated: a lack of algorithm *comparability* and a lack of *repeatability* regarding experiments. Algorithms are usually tailored to their specific PDES system, and experiments are usually done using *specific* hardware to simulate *specific* application models with *specific* features (see table 6 in [58]). These problems lead to incomplete knowledge about the advantages and disadvantages of PDES algorithms, which in turn causes high development efforts and uncertain results. This was described in chapter 3.

Besides that, different ways of overcoming these issues were discussed. It was argued that PDES systems may exhibit *complex* behaviour, which is caused by the complexity of the models they simulate. This would also explain why analytical analysis of PDES algorithms is so hard (see section 3.1). These considerations led to the conclusion that modelling and simulation techniques, which are often used to analyse complex systems, can be successfully applied in this context. This is the core assumption of this work and therefore had to be confirmed.

To do so, chapter 4 detailed a general purpose simulation simulator, SimSim, and its implementation. The implementation fulfils the proposed requirements of section 4.2: SimSim possesses an interactive mode as well as a batch mode, it exposes interfaces for plug-ins and supports the plug-in developer with means to generate suitable visualisations and custom parameterisation dialogues with ease. Its architecture facilitates simulation observation and the coupling with external tools (e.g. network simulators).

SimSim was used as a development aid during the creation of a load balancing algorithm for James II's abstract PDEVS simulator (chapter 5). Afterwards, the developed algorithm was thoroughly tested to show the strengths of such an approach for PDES algorithm development (chapter 6). Figure 7.1 illustrates how the considerations from section 4.1 have been implemented to proof the overall concept (as described in section 3.5). For the given example, the experiments yield the identification of:

- *Disadvantageous mechanisms*: Section 6.2.1 and 6.2.4 indicate that the adaptation mechanism worked out in section 5.4.3 does *not* contribute to the algorithm's performance. Interactive SimSim runs were used to identify the problems that hamper its efficiency (section 6.2.5).
- *Advantageous mechanisms*: Sections 6.2.1 and 6.2.4 also indicate that considering migration cost and using the voting scheme (both described in section 5.4.2) are indeed beneficial to the overall performance. However, it was also shown that the benefits are particularly large in a region of the simulation space that is rather uninteresting for load balancing algorithms, because the granularity is too low for efficient distributed simulation.

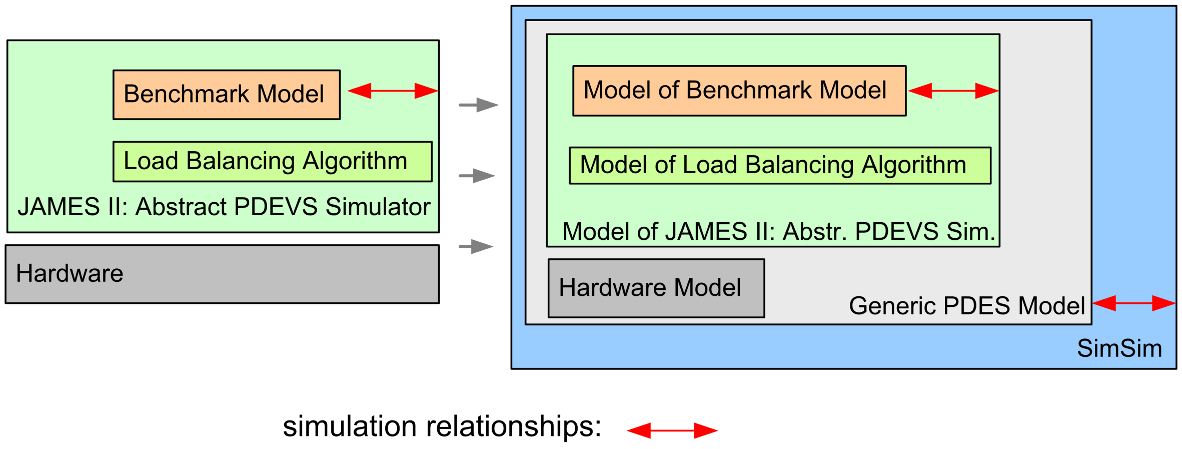

Benchmark Model
Load Balancing Algorithm
JAMES II: Abstract PDEVS Simulator
Hardware
Model of Benchmark Model
Model of Load Balancing Algorithm
Model of JAMES II: Abstr. PDEVS Sim.
Hardware Model
Generic PDES Model
SimSim
simulation relationships:

Figure 7.1: Overview of Implemented Components. This illustration shows how the basic components of a simulation simulator (see figure 4.1 on page 45) have been realised. The colours correspond to the same scheme that was used for figure 4.3. Additionally, hardware and load balancing components were highlighted.

- *Sensitive parameters*: The effect of various parameters, such as inherent parallelism, number of LPs, migration cost, or message transfer time, has been investigated. This allows to identify particularly important parameters for certain PDES system requirements. For example, the results in section 6.2.3 indicate the importance of tree size, branch factor and inherent parallelism with respect to the scalability.
- *Insensitive parameters*: Section 6.2.4 illustrated that *insensitive* parameters can also be identified. Although their insensitivity may only hold for a very small region of the simulation space, this exemplifies how a prior simulation may help to rule out aspects that one would otherwise have to worry about during development.

Section 6.3 also investigated the accuracy of SimSim's predictions. Except for low granularity scenarios and situations in which many busy threads are executed by one CPU, the results were found to be acceptable (approximately 10% deviation from real performance).

All in all, it can be said that the assumption of using modelling and simulation techniques in this context is underpinned by the obtained results of the 'test case' detailed in chapter 5 and 6. SimSim was useful to evaluate the gain of different mechanisms that may be part of the load balancing algorithm, like the voting scheme or the consideration of migration cost. It was possible to assess the quality of each mechanism under various conditions, rather than implementing it in James II and testing it by hand. With SimSim, not only the flaws of certain approaches could be identified easily (e.g. too optimistic gain estimation in section 6.2.1, oscillating adaptation in section 6.2.5), but also promising future enhancements to focus on. Since some gain region borders could be determined for each setup of the algorithm, it was also possible to find out the regions in which the algorithm's behaviour ought to be enhanced.

Furthermore, it was demonstrated how the modelling of a concrete PDES system in SimSim facilitates certain development tasks. For example, the prototype implementation of the load balancing algorithm does not have to care about the accuracy of its observations – the observation approaches were simply abstracted away, their accuracy was assumed. This allows a fast and continuous testing and fits nicely to state-of-the-art programming techniques like *extreme programming* (see section 7.3).

However, neither SimSim nor the general approach of simulating simulations is a silver bullet for PDES development. One has to accept additional efforts induced by firstly developing a *valid* model of all relevant PDES system parts, and secondly by modelling the algorithm and appropriate benchmark models. These problems and possible solutions are detailed in the next sections.

7.2 Putting the Work in Context

Basically, the development of a PDES algorithm within an PDES system model can be regarded as an application of the fundamental *"divide and conquer"* principle to the development of PDES algorithms. The problem of developing a suitable PDES algorithm to solve a particular problem is divided into two separate tasks that can be solved independently: The first task is to find algorithmic methods that solve the given problem, so to say its 'theoretical' nature. The second task is the *concrete* implementation of this algorithm, usually specific to a programming language, an operating system, and a set of already existing components.

Splitting up both tasks brings several advantages. At first, the development of possibly large and sophisticated algorithms can be facilitated by using SimSim in the same way as described in chapter 5, i.e. by analysing individual runs. If the PDES system consists of different mechanisms that might *influence* each other, or if the application models differ strongly, a thorough analysis of a prototype – as described in chapter 6 for the load balancing algorithm – might be *very* important as well. How much impact some rather 'small' changes on an algorithm's overall performance might have was exemplified by testing the adaptation mechanism. So, the first task of carving out a proper algorithm can be greatly facilitated by prior simulation.

Many constraints of the second task may be abstracted away when developing the algorithm (e.g. programming language, communication layer, etc.), but even so its functioning is already known when it comes to its final implementation. This allows the developer to concentrate on the implementation details, which might be quite tricky and difficult to solve as well, instead of focusing on the way the algorithm works. In other words, I propose a *layered* approach to develop PDES algorithms. From yet another perspective, one could simply regard it as testing on a more abstract level, using a kind of PDES algorithm sandbox.

Although both tasks are independent from each other *in theory*, it might happen that an algorithm's predicted performance is quite different from reality. This happens when the specific system model or application model is *invalid* with respect to the problem to be investigated. As described in section 4.3 and 4.4, SimSim was designed in a way that prevents very crude violations of validity by letting parts of a distributed algorithm communicate via simulated messages. This level of detail can ensure that *all* information exchanged by LPs is transmitted via messages[1]. If a message-based protocol works under these circumstances, it will most likely do so in reality. However, this does *not* mean that more subtle inconsistencies to the real world make the model invalid nevertheless. Unfortunately, model validation is quite a hard problem that induces additional development cost.

One way to overcome this problem is to integrate the application of a simulation simulator into the software-lifecycle. In most cases, the development of a complex software system incorporates *multiple* development cycles from requirements identification to testing a software prototype. This way of developing software – as opposed to sequentially executing requirements identification, specification, implementation, and testing – is particularly endorsed by extreme programming, which became rather popular in the last years [67]. When integrating the development of a PDES system model with the development of the PDES system, the former could already be tested against the real-world prototype at an early development stage. At the same time, simulations using the up-to-date and tested PDES system model would yield new insights and facilitate the generation of the next prototype. The simulation approach is more efficient than testing the prototype in a simpler way, not just because different constraints are abstracted away, but because the tests can be executed on a single computer, and do not even require the existence of the environment on which the real PDES system should run in the future.

Having said that, it has to be stressed that the trade-off between additional development efforts for a PDES system model and the beneficial effects on the overall development process depends largely on the complexity of the task at hand. So, a prior simulation of a rather simple algorithm, which is designed for a PDES system whose behaviour is well-known, might be much more laborious than implementing it directly.

[1]Of course, these precautions can be easily circumvented by the plug-in developer, which facilitates the development of certain auxiliary mechanisms. Anyhow, these circumventions have to be done *deliberately*, and I optimistically assume that the developer knows about the consequences.

Indeed, this is an aspect that applies to all abstract means to facilitate software development, like UML. No one would use a UML class diagram to model two classes. But when it comes to hundreds of classes, a class diagram may prove to be quite useful. Actually, the idea of using UML and *model-driven* software engineering is closely related to the simulation approach proposed in this work.

In a recent issue of the IEEE *Computer* magazine covering model-driven engineering, Gray et al. point out that *"Designers must be able to examine various design alternatives quickly and easily among myriad and diverse configuration possibilities. Ideally, a tool would simulate each new design configuration so that designers could rapidly determine how some configuration aspect, such as a communication protocol, affects an observed property, such as throughput."*[41, p. 51]. Furthermore, France et al. state that *"The ability to animate models can help developers better understand modeled behavior, while testing models can help uncover errors before developers transform the models into implementations. Both novices and experienced developers will benefit from the visualization of modeled behavior that model animators provide."*[33, p. 63 - 64] in the same issue. Clearly, this is the same argumentation used to motivate the interactive mode of SimSim in section 4.2. Of course, the quoted statements have been made in the context of software engineering, and specifically with respect to the rather new UML 2.0 standard. Nevertheless, since the development of PDES systems *is* software development, SimSim addresses the same problems, albeit in a much more specific and not so sophisticated way. Model-driven engineering is just the other side of the same coin.

Finally, it has to be mentioned that similar approaches exist to simulate distributed systems in general. For example, Buyya et al. present GridSim to simulate the behaviour of grid software [15], and Adve et al. propose a framework that combines analysis and simulation tools to predict the performance of "large parallel adaptive computational systems" in [3]. Actually, James II and PDES-MAS fall into a similar category, since both systems were designed to simulate multi-agent systems. Multi-agent systems can be regarded as specific distributed systems, and simulation tools are used to facilitate their development. After all, it can be seen that the *general* idea of using simulation and modelling for software development is anything but new.

7.3 Outlook and Future Work

Although the results that were summed up in section 7.1 generally encourage the integration of simulation with the development process of PDES algorithms, there are still many open questions. First of all, it would be interesting to investigate how the efforts of model-driven engineering can be integrated with simulating PDES systems. For example, State Charts could be used to describe an algorithm's behaviour, but there are many other formalisms that could be used for this purpose as well. It might even be favourable to derive an own modelling formalism to model PDES systems conveniently. A standardisation could also enable to re-use application models, so that PDES algorithms of all kinds could be compared more objectively. This would also enhance the repeatability of PDES experiments, so that the two problems named at the end of chapter 2 could be alleviated. The assessment of PDES algorithms on such an abstract level will yield results that are unbiased with respect to programming language, implementation quality, or hardware. On the other hand, an algorithm's real performance is also influenced by factors that are *not* modelled (e.g. computing time overheads, memory consumption, etc.). Therefore, simulation systems are still needed for *concrete* comparisons, since they allow to get a more accurate picture of these factors.

Furthermore, the result analysis needs to be improved dramatically, so that this approach is indeed able to explore the relevant simulation space for a given PDES system and simulation problem. Note that the simulation problem to prove my point consisted of *one* simulation algorithm and *one* load balancing algorithm, leaving out any external process handling or additional synchronisation schemes. Moreover, very restrictive assumptions had to be made, otherwise there would have been too much parameters to be investigated. Still, there were many different parameters and scenarios to be analysed manually, which caused a considerable amount of extra work.

In [108], Zhou et al. point out in a similar context that "*'because of the large number of options for each component policy, it is impossible to study all possible policy combinations in this research."'*[108, p.

1330]. While this is true when running experiments manually, an *automated* simulation parameterisation, combined with machine learning mechanisms or data mining methods, could greatly facilitate this task. Moreover, running a simulation with all parameter combinations in a certain interval is a rather brute-force method. Additional mechanisms could be used to avoid evaluating all combinations. This would allow a faster analysis of a simulation space region. As could be seen in chapter 6, most performance curves seemed to be rather smooth, which could motivate the usage of mathematical interpolation and extrapolation techniques.

Thinking this approach out, one could use the knowledge gained by these methods to compile the best known PDES systems for a given simulation problem, i.e. a given environment and a given model with certain properties (e.g. granularity, dynamism, etc.). Another idea would be to let an agent control the PDES execution, and letting it swap and tweak the used algorithms at runtime. Its knowledge about the specific peculiarities of the controlled PDES system could be obtained by simulating the system beforehand. The agent could even run a symbiotic simulation of a PDES system, in order to test possible parameter adjustments for the concurrently executing PDES system at runtime.

To realise such a system, mathematical optimisation methods could be used as well, since the agent would actually perform a local search in the multidimensional simulation space. The execution time as predicted by the simulation of a PDES system would represent the approximated cost to be minimised. The *real* cost function, which determines the *real* simulation performance and was defined as f_perf in section 3.4, would be approximated via prediction. Of course, a PDES system to be extended by such mechanisms would have to be sufficiently flexible and component-oriented to allow these modifications by an external component at runtime.

Furthermore, SimSim could be used for educational purposes in computer science. Using the event list and a proper visualisation, debugging and testing of distributed algorithms simulated by SimSim is relatively simple, so that the software could be used to teach PDES at the university. All students could, for example, program different GVT calculation algorithms and test them at home under various conditions, without needing any additional hardware.

However, there are also more profane tasks to be done in the future: While SimSim's core features are completely implemented, the user interface, the visualisation and the implementation of the event queue could be enhanced. Besides that, it would be interesting to quantify the effect of using a real network simulator instead of the communication cost model as described in section 4.4. SimSim could also be integrated more closely with James II, especially with respect to the handling of XML parameter files and the user interface. Finally, section 6.3 motivates the enhancement of the generic PDES system model, so that the the prediction quality is increased.

Regarding the load balancing algorithm, I have to concede that additional efforts are necessary, both on algorithmic and on implementation level, until it is more than just a prototype. Possible enhancements have already been identified in section 5.5.

Bibliography

[1] *The American Heritage Dictionary*. Bantam Dell, 2004.

[2] Java SciMark 2.0. http://math.nist.gov/scimark2/.

[3] Vikram S. Adve, Rajive Bagrodia, James C. Browne, Ewa Deelman, Aditya Dube, Elias N. Houstis, John R. Rice, Rizos Sakellariou, David Sundaram-Stukel, Patricia J. Teller, and Mary K. Vernon. POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems. *IEEE Trans. Software Eng*, 26(11):1027–1048, 2000.

[4] Ahmad Afsahi and Nikitas J. Dimopoulos. Hiding communication latency in reconfigurable message-passing environments. In *Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, IPPS/SPDP'99 (San Juan, Puerto Rico, April 12-16, 1999)*, pages 55–60, Los Alamitos-Washington-Brussels-Tokyo, 1999. IEEE Computer Society, ACM SIGARCH, IEEE Computer Society Press.

[5] Heidi R. Ammerlahn, Michael E. Goldsby, Michael M. Johnson, and David M. Nicol. A geographically distributed enterprise simulation system. *Future Gener. Comput. Syst.*, 17(2):135–146, 2000.

[6] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A Parallel Simulation Environment for Complex Systems. *Computer*, 31(10):77–85, 1998.

[7] Rajive Bagrodia and Mineo Takai. Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages. *IEEE Trans. Parallel Distrib. Syst*, 11(4):395–411, 2000.

[8] Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A Language for the Design of Efficient Discrete-Event Simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, April 1994.

[9] Vijay Balakrishnan, Peter Frey, Nael B. Abu-Ghazaleh, and Philip A. Wilsey. A Framework for Performance Analysis of Parallel Discrete Event Simulators. In *Winter Simulation Conference*, pages 429–436, 1997.

[10] Vijay Balakrishnan, Radharamanan Radhakrishnan, Dhananjai Madhavarao, Nael Abu-Ghazaleh, and Philip A. Wilsey. A Performance and Scalability Analysis Framework for Parallel Discrete Event Simulators.

[11] Ioana Banicescu and Rong Lu. Experiences With Fractiling In N-Body Simulations, April 28 1998.

[12] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. Comput. Syst.*, 13(1):1–31, 1995.

[13] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[14] Azzedine Boukerche. An Adaptive Partitioning Algorithm for Conservative Parallel Simulation. In *IPDPS*, page 133, 2001.

[15] Rajkumar Buyya and Manzur 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.

[16] Junwei Cao, Daniel P. Spooner, Stephen A. Jarvis, Subhash Saini, and Graham R. Nudd. Agent-Based Grid Load Balancing Using Performance-Driven Task Scheduling. In *17th International Parallel and Distributed Processing Symposium (IPDPS-2003)*, pages 49–49, Los Alamitos, CA, April 22–26 2003. IEEE Computer Society.

[17] Christopher D. Carothers and Richard 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.

[18] Christopher D. Carothers, Brad Topol, Richard M. Fujimoto, John T. Stasko, and Vaidy S. Sunderam. Visualizing Parallel Simulations in Network Computing Environments: A Case Study. In *Winter Simulation Conference*, pages 110–117, 1997.

[19] John S. Carson. Modeling and Simulation Worldviews. In *Winter Simulation Conference (WSC '93)*, pages 18–23, New York, December 1993. ACM Association for Computing Machinery.

[20] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *WSC '94: Proceedings of the 26th conference on Winter simulation*, pages 716–722, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[21] James H. Cowie. *Scalable Simulation Framework API Reference Manual*.

[22] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The Department of Defense High Level Architecture. In *WSC '97: Proceedings of the 29th conference on Winter simulation*, pages 142–149, New York, NY, USA, 1997. ACM Press.

[23] Samir R. Das. Adaptive protocols for parallel discrete event simulation. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 186–193, New York, NY, USA, 1996. ACM Press.

[24] Samir R. Das, Richard M. Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Winter Simulation Conference*, Proceedings of the 1994 Winter Simulation Conference, pages 1332 – 1339, 1994.

[25] Daniela Degenring, Mathias Röhl, and Adelinde M. Uhrmacher. Discrete Event Simulation for a Better Understanding of Metabolite Channeling - A System Theoretic Approach. In *CMSB*, pages 114–126, 2003.

[26] Fred Douglis and Ian T. Foster. IEEE Internet Computing: Guest Editors' Introduction - The Grid Grows Up. *IEEE Distributed Systems Online*, 4(7), 2003.

[27] Bruce Edmonds. What is Complexity? - The philosophy of complexity per se with application to some examples in evolution, 1999.

[28] David Eppstein. Small Maximal Independent Sets and Faster Exact Graph Coloring. *J. Graph Algorithms & Applications*, 7(2):131–140, 2003.

[29] Roland Ewald. Modellpartitionierung in JAMES II. 2005.

[30] Roland Ewald, Dan Chen, Georgios K. Theodoropoulos, Michael Lees, Brian Logan, Ton Oguara, and Adelinde M. Uhrmacher. Performance Analysis of Shared Data Access Algorithms for Distributed Simulation of Multi-Agent Systems. In *Proceedings of the 20th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2006)*, 2006.

[31] Per-Olof Fjällström. Algorithms for graph partitioning: A Survey. In *Linkoping Electronic Atricles in Computer and Information Science, 3.*, 1998.

[32] Poseidon for UML. http://gentleware.com/.

[33] Robert B. France, Sudipto Ghosh, and Trung Dinh-Trong. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *Computer magazine*, pages 59 – 66, February 2006.

[34] R. M. Fujimoto. Time Warp on a Shared Memory Multiprocessor. *Transactions of the Society for Computer Simulation Intl.*, 6(3):211–239, July 1989.

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

[36] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley, 2000.

[37] Richard M. Fujimoto, Samir R. Das, Kiran S. Panesar, Maria Hybinette, and Chris Carothers. *Georgia Tech Time Warp (GTW Version 3.1) Programmer's Manual for Distributed Network of Workstations*, 1997.

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[39] Martina Gierke and Adelinde M. Uhrmacher. Modeling Elderly Behavior for Simulation-based Testing on Agent Software. In *Proc. of the Conceptual Modeling and Simulation Cocnference (CMS 2005)*, 2005.

[40] Ezequiel Glinsky and Gabriel Wainer. Performance Analysis of DEVS Environments. In *Proceedings of AI Simulation and Planning*, 2002.

[41] Jeff Gray, Yuehua Lin, and Jing Zhang. Automating Change Evolution in Model-Driven Engineering. *Computer magazine*, pages 51 – 58, February 2006.

[42] Jordi Guitart, Jordi Torres, Eduard Ayguade, and J. M. Bull. Performance Analysis Tools for Parallel Java Applications on Shared-memory Systems. In *Proceedings of 2001 International Conference on Parallel Processing (30th ICPP'01)*, Valencia, Spain, September 2001. Universidad Politecnia de Valencia.

[43] 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, October 1991.

[44] Fang Hao, Karen Wilson, Richard M. Fujimoto, and Ellen Zegura. Logical process size in parallel simulations. In *WSC '96: Proceedings of the 28th conference on Winter simulation*, pages 645–652, New York, NY, USA, 1996. ACM Press.

[45] Jan Himmelspach and Adelinde 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.

[46] Soonwook Hwang and Carl Kesselman. GridWorkflow: A Flexible Failure Handling Framework for the Grid. In *HPDC*, pages 126–137, 2003.

[47] Eclipse IDE. http://www.eclipse.org/.

[48] Java. http://java.sun.com/.

[49] David Jefferson, Brian Beckman, Fred Wieland, Les Blume, Mike DiLoreto, Phil Hontabas, Prine Laroche, Kathy Sturdevant, Jack Tupman, Van Warren, John Wedel, Herb Younger, and Steve Bellenot. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (11th SOSP11), ACM Operating Systems Review (OSR)*, volume 20, pages 77–93, Austin Texas, 1987.

[50] Ming Jiang, Georgios K. Theodoropoulos, and Rachid Anane. A framework for distributed simulation on a grid. In *WISICT '04: Proceedings of the winter international synposium on Information and communication technologies*. Trinity College Dublin, 2004.

[51] Zoltan Juhasz, Stephen Turner, Krisztian Kuntner, and Miklos Gerzson. A Performance Analyser And Prediction Tool For Parallel Discrete Event Simulation. In *UKSIM 2001: Conference On Computer Simulation*, 2001.

[52] Zoltan Juhasz, Stephen Turner, Krisztian Kuntner, and Miklos Gerzson. A Trace-based Performance Prediction Tool for Parallel Discrete Event Simulation. In *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2002*, pages 338 – 344, 2002.

[53] Kihyung Kim, Wonseok Kang, Bong Sagong, and Hyungon Seo. Efficient Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-Hierarchical One. In *Annual Simulation Symposium*, pages 227–233, 2000.

[54] Bora I. Kumova. Platform Extensions of a Distributed Simulation Kernel. In *2004 High Peformance Computing Symposium*, pages 67 – 70, 2004.

[55] Michael Lees, Brian Logan, and Georgios K. Theodoropoulos. Time windows in multi-agent distributed simulation. In *Proceedings of the 5th EUROSIM Congress on Modelling and Simulation (EuroSim'04)*, 2004.

[56] Jason Liu, David M. Nicol, Brian J. Premore, and Anna L. Poplawski. Performance Prediction of a Parallel Simulator. In *Workshop on Parallel and Distributed Simulation*, pages 156–164, 1999.

[57] Brian Logan and Georgios 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.

[58] Y. Low, C. Lim, W. Cai, S. Huang, W. Hsu, S. Jain, and S. Turner. Survey of languages and runtime libraries for parallel discrete-event simulation. *Simulation*, pages 170–186, March 1999.

[59] Alke Martens and Jan Himmelspach. Combining Intelligent Tutoring and Simulation Systems. In *Proc. of the International Conference on Human-Computer Interface Advances for Modeling and Simulation SIMCHI'05*, pages 65–70, 2005.

[60] Dale E. Martin, Timothy J. McBrayer, and Philip A. Wilsey. WARPED: a time warp simulation kernel for analysis and application development. In H. El-Rewini and B. D. Shriver, editors, *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, volume 1, pages 383–386, 1996.

[61] Dale E. Martin, Philip A. Wilsey, Robert J. Hoekstra, Eric R. Keiter, Scott A. Hutchinson, Thomas V. Russo, and Lon J. Waters. Redesigning the WARPED Simulation Kernel for Analysis and Application Development. In *Annual Simulation Symposium*, pages 216–223, 2003.

[62] Friedemann Mattern. Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, August 1993.

[63] Laurent Michel and Pascal Van Hentenryck. Maintaining Longest Paths Incrementally, February 04 2003.

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

[65] Pieter J. Mosterman. An Overview of Hybrid Simulation Phenomena and Their Support by Simulation Packages. In *HSCC*, pages 165–177, 1999.

[66] mpC Parallel Programming Environment. http://www.ispras.ru/∼mpc/.

[67] James Newkirk. Introduction to agile processes and extreme programming. In *ICSE*, pages 695–696, 2002.

[68] David M. Nicol. Scalability, locality, partitioning and synchronization PDES. In *Proceedings of the twelfth workshop on Parallel and distributed simulation*, pages 5–11. IEEE Computer Society, 1998.

[69] David M. Nicol. Utility Analysis of Parallel Simulation. In *PADS '03: Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 123, Washington, DC, USA, 2003. IEEE Computer Society.

[70] David M. Nicol, Michael M. Johnson, Ann S. Yoshimura, and Michael E. Goldsby. Performance Modeling of the IDES Framework. In *Workshop on Parallel and Distributed Simulation*, pages 38–45, 1997.

[71] David M. Nicol, Michael M. Johnson, Ann S. Yoshimura, and Michael E. Goldsby. IDES: A Java-based Distributed Simulation Engine. In *Proceedings of the MASCOTS*, pages 233–240, 1998.

[72] David M. Nicol, Jason Liu, Michael Liljenstam, and Guanhua Yan. Simulation of large scale networks I: simulation of large-scale networks using SSF. In *Winter Simulation Conference*, pages 650–657, 2003.

[73] Ton Oguara, Dan Chen, Georgios K. Theodoropoulos, Brian Logan, and Michael Lees. An Adaptive Load Management Mechanism for Distributed Simulation of Multi-agent Systems. In *DS-RT*, pages 179–186, 2005.

[74] Vern Paxson. End-to-End Routing Behavior in the internet. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26,4 of *ACM SIGCOMM Computer Communication Review*, pages 25–39, New York, August 26–30 1996. ACM Press.

[75] Kalyan S. Perumalla, Matthew Andrews, and Sandeep Bhatt. A virtual PNNI Network Testbed. In *Proceedings of the 1997 Winter Simulation Conference*, 1997.

[76] Kalyan S. Perumalla, Matthew Andrews, and Sandeep N. Bhatt. TED Models for ATM Internetworks. *SIGMETRICS Performance Evaluation Review*, 25(4):12–21, 1998.

[77] Kalyan S. Perumalla and Richard M. Fujimoto. Interactive parallel simulations with the JANE framework. *Future Generation Computer Systems*, 17(5):525–537, March 2001.

[78] Kalyan S. Perumalla, Richard M. Fujimoto, Prashant J. Thakare, Santosh Pande, Homa Karimabadi, Yuri Omelchenko, and Jonathan Driscoll. Performance Prediction of Large-Scale Parallel Discrete Event Models of Physical Systems. In *Winter Simulation Conference 2005*, 2005.

[79] Eclipse Metrics plug in. http://sourceforge.net/projects/metrics.

[80] Martha E. Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. pages 183–189, 1990.

[81] B. Preiss, W. Loucks, and V. Hamacher. A Unified Modeling Methodology for Performance Evaluation of Distributed Discrete Event Simulation Mechanism. In *Proceedings of the Winter Simulation Conference*, pages 315–324, 1988.

[82] B. Preiss and I. Macintyre. YADDES - Yet Another Distributed Discrete Event Simulator: User Manual. Technical Report CCNG E-197, University of Waterloo, Waterloo, 1990.

[83] Krithi Ramamritham and John A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, 1994.

[84] Luís Paulo Reis and Nuno Lau. FC Portugal Team Description: RoboCup 2000 Simulation League Champion. In *RoboCup*, pages 29–40, 2000.

[85] Craig Reynolds. Boids. http://www.red3d.com/cwr/boids/.

[86] Mathias Röhl and Adelinde M. Uhrmacher. Flexible Integration of XML Into Modeling and Simulation Systems. In *Proc. of the Winter Simulation Conference*, 2005.

[87] Robert Rosen. *Essays on Life Itself*. Columbia University Press, 1999.

[88] Heinz G. Schuster. *Complex Adaptive Systems*. Scator Verlag, 2001.

[89] SSF Network Simulation. http://www.ssfnet.org.

[90] Milan E. Soklic. Simulation of load balancing algorithms: a comparative study. *SIGCSE Bull.*, 34(4):138–141, 2002.

[91] Lisa M. Sokol and Brian K. Stucky. MTW: Experimental Results for a Constrained Optimistic Scheduling Paradigm. In David Nicol, editor, *Distributed Simulation*, volume 22 of *Simulation*, pages 169–173. Society for Computer Simulation (SCS), San Diego, CA, January 1990.

[92] XML Specification. http://www.w3.org/TR/REC-xml/.

[93] Jeff S. Steinman. Interactive SPEEDES. In *Annual Simulation Symposium*, pages 149–158, 1991.

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

[95] Jeff S. Steinman. Breathing Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation (PADS '93)*, 1993.

[96] Swaminathan Subramanian, Dhananjai M. Rao, and Philip A. Wilsey. Applying Multilevel Partitioning to Parallel Logic Simulation. *Parallel and Distributed Computing Practices*, 4(1), 2001.

[97] Peihan Teo, Stephen John Turner, and Zoltan Juhasz. Optimistic Protocol Analysis in a Performance Analyzer and Prediction Tool. In *PADS*, pages 49–58, 2005.

[98] Batik SVG Toolkit. http://xml.apache.org/batik/.

[99] Alejandro Troccoli and Gabriel Wainer. Performance Analysis of Cellular Models with Parallel Cell-DEVS. In *2001 Summer Computer Simulation Conference (SCSC)*, pages 68–73, 2001.

[100] A. M. Uhrmacher and K. Gugler. Distributed, parallel simulation of multiple, deliberative agents. In *PADS '00: Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pages 101–108, Washington, DC, USA, 2000. IEEE Computer Society.

[101] Federick Wieland, Eric Blair, and Tony Zukas. Parallel Discrete-Event Simulation (PDES): A Case Study in Design, Development, and Performance Using SPEEDES. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation (PADS '95)*, pages 103–110, 1995.

[102] Linda F. Wilson and David M. Nicol. Automated Load Balancing in SPEEDES. In *Winter Simulation Conference*, pages 590–596, 1995.

[103] Linda F. Wilson and David M. Nicol. Experiments in Automated Load Balancing. In *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation*, Washinton, May 22–24 1996. IEEE Computer Society Press.

[104] Paul Wonnacott and David Bruce. A Prototype Implementation of APOSTLE and its Performance. In *Proceedings of the l995 SCS Summer Computer Simulation Conference*, pages 197 – 205, 1995.

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

[106] Bernard P. Zeigler. *Multifacetted modelling and discrete event simulation*. Academic Press, New York, 1984.

[107] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation, Second Edition*. Academic Press, 2000.

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

[109] Hubert Zimmermann. OSI reference model - The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.

Glossary

Abstract Class

Abstract classes are classes that *cannot* be instantiated. They reflect concepts that are too general to be represented by concrete objects. For example, a class `Vehicle` should be abstract, since there is no such thing as a vehicle, apart from being a general term that includes cars, bikes, or trains. Therefore, a (non-abstract) `Car` class could inherit from `Vehicle` to express the relationship that cars *are* vehicles. Abstract classes are often used as placeholders, so that the polymorphism of an object-oriented language can be exploited.

ATM Switch

A switch is a network component that serves as a connector in a network. It connects computers and other components of a network (e.g. other switches). Asynchronous transfer mode (ATM) is a standard network protocol for layer two of the OSI reference model.

Cellular Automata

Cellular Automata (CA) can be used to model systems that consist of elements whose behaviour depends only on the state of the surrounding. Each automaton consists of a number of cells, upon which a neighbourhood relation is defined. CA are usually simulated using discrete time steps. The next state of each cell is defined by a function of it's neighbours current states.

Client/Server Architecture

In the domain of software systems, a client/server architecture allows multiple processes (the clients) to use a centralised software service (the server). This technique is often used for running software in large networks. For example, the internet basically consists of servers (e.g. web, mail, streaming) that provide content for the millions of clients.

Cluster

A cluster is a number of (usually homogeneous) computers that are are connected via a relatively fast local area network connection. They can be virtualised, so that they form a single logical computer. Because of their comparably low costs, clusters are often used as a replacement for supercomputers.

CPU

The central processing unit (CPU), also called processor, is the decisive resource that determines a computer's speed with respect to mathematical computations.

Depth First Search (DFS)

Depth first search is a recursive search technique that starts at a certain element and then applies itself on a 'neighbour' element, until a solution is found. It returns if no solution is found and there is no neighbour element left that can be analysed. Hence, it resembles to traversing a 'tree' of possible solutions. Combined with a mechanism to detect circles, DFS can also be used for colouring arbitrary graphs.

Design Pattern

Design patterns, or simply patterns, are commonly known and applied techniques to avoid certain problems that arise in more complex software systems. Patterns are widely used to describe the structural composition of a software system.

Event Queue

An event queue is a data structure that holds the (unprocessed) events of a simulation component.

Extreme Programming

Extreme Programming is a software development methodology that particularly addresses the inherent dynamism of software projects, e.g. unforeseen specification and requirements changes. To alleviate these problems, a continuous prototyping of the software is one of the key development techniques of extreme programming [67].

Garbage Collection

Garbage collection is a mechanism that frees memory by removing unnecessary variables. When a large number of objects is stored in the memory, the identification of objects that can be removed might be quite time-consuming.

Global Virtual Time (GVT)

The global virtual time is the minimal event time stamp of all LPs of a PDES. For example, the GVT computation is used for optimistic simulation, so that information that is not needed anymore can be removed by the *garbage collection*.

Grid

In computer science, a grid refers to computers that are virtualised, so that say can be used as a single computer, similar to *clusters*. In contrast to clusters, grids consist of *inhomogeneous* computer hardware which may be geographically distributed. This leads to some complications. For example, network latencies in a grid environment can be relatively large.

IDE

Integrated Development Environments (IDEs) are complex development tools that integrate editors, compilers, debuggers, and other programming tools to facilitate programming.

Java Bean

A Java Bean is a Java object that complies with the Java Bean specification. For example, a bean object has to consist of serialisable data structures (i.e. data structures that can be expressed as a string) and implement functions to set and retrieve its fields. These have to follow the Java naming conventions (`getXYZ(...)`, `setXYZ(...)`). Java Beans are a standard to support component-based software design.

Java Virtual Machine

A Java Virtual Machine (JVM) is basically a software that interprets Java bytecode (i.e. `*.class` files). A JVM has to follow a certain specification, but may introduce mechanisms to optimise the bytecode execution, for example by using a just-in-time compiler.

Landau Notation

The Landau notation is used to define the time and space complexity of algorithms. The Big-Oh notation used in this work, which is only a part of the Landau notation, is the most common notation for complexity in computer science. If an algorithm needs $O(g(n))$ time for its execution, this means that $g(n)$ is the minimal function for which there is a constant k, so that the exact execution time function $f(n)$ is *always* less than $k \cdot g(n)$. Hence, the Big-Oh notation can be regarded as a worst-case approximation.

Lookahead

The lookahead of an LP is a time interval in simulation time for which it is sure that the LP will not generate any new events. Similarly, a lookahead can be defined with respect to single LPs. It is often used by conservative synchronisation approaches to exploit parallelism.

Message Passing Interface (MPI)

The message passing interface (MPI) is a protocol that serves for inter-process communication. It is often used by PDES systems as a communication layer.

Modelling Formalism

A modelling formalism is a formal method to specify models. Examples for modelling formalisms are DEVS, PDEVS, or *cellular automata*.

Multi-Agent System (MAS)

A Multi-Agent System (MAS) is a system that incorporates multiple agents. Here, agents are used as a metaphor for heavy weight objects that show complex behaviour. Often, agents employ artificial intelligence methods for reasoning and therefore have high computing demands. Additionally, they might manage large states. This makes developing MAS quite challenging, so that PDES systems are often used to analyse their behaviour. Examples for MAS are Tileworld [80] or Boids [85].

Network Hop

A network hop is the transfer of a message between two nodes in a network. Routing protocols aim at minimising the number of network hops a message needs to get from its source to its destination.

NP-hard

In short, NP-hard problems are problems that are at least as hard to solve as *any* problem in NP. NP is the class of problems that can be solved by a *non-deterministic* Turing machine in polynomial time. Since today's computers only have the capabilities of *deterministic* Turing machines, NP-hard problems require at least exponential time when running on them (that is, unless $P = NP$ could be shown).

OSI Reference Model

The Open Systems Interconnection Reference Model, also named (ISO-)OSI Reference Model, is a central communication standard. It defines different layers that are involved in electronic communication. Then, it precisely describes which communication problem will be resolved by which layer in the system. This results in a modular structure that allows to freely combine different components that manage distinct OSI layers.

P2P System

P2P (or Peer-to-Peer) systems are distributed systems with a flat hierarchy. Each node may act as a server or a client when interacting with other nodes. Peer-to-peer systems are widely used to realise completely decentralised systems, e.g. in the domain of file sharing.

Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM) software package allows the virtualisation of a computer network. This means, by running PVM as middleware (i.e. as an additional software layer between operating system and application software), the programmer can write PVM-based programs as if they were executed on a single computer.

PHOLD Model

The parallel HOLD model (PHOLD) was developed by Fujimoto to analyse the performance of time warp [35]. In a PHOLD model, messages are circulating between LPs. The routing of the messages, the execution time each message evokes, and other parameters can be configured by stochastic distributions. For further details, refer to [35].

Plug-in

A plug-in is a software component that enhances an existing software system, without causing it to be re-compiled or altered in any another way. Basically, plug-ins are simply 'plugged' into software systems via a pre-defined interface. This ensures a clean separation of a software's core functionality and possible extensions.

Polymorphism

Polymorphism, also called late binding, is a core feature of object-oriented programming languages. It allows the use of super classes (e.g. *abstract classes*) instead of their sub-classes. Using polymorphism allows to define code that is able to cope with instances of *any* sub-class that inherits from the super class. This makes it possible to write code whose behaviour will not be defined until runtime, when the actual type of the objects it works with is known. This contributes to the flexibility and re-usability of object-oriented program code.

Remote Method Invocation (RMI)

The Remote Method Invocation (RMI) functionality of Java provides support for inter-process communication. Therefore, it is an alternative for MPI when developing distributed Java programs.

Routing Algorithm

Routing algorithms are used in networks to route messages from their source to their destination. Usually, routing algorithms try to minimise the time it takes for a message to reach its destination, or the number of *network hops* a message needs.

Software Design Pattern

See description of *design pattern*.

State Charts

State Charts are diagrams that allow the definition of an object's behaviour. Basically, state charts are directed graphs in which each node describes a state of the object. Correspondingly, the edges of this graph describe the state transitions that may occur. Edges and nodes can now be labelled with various additional information. Moreover, there are notations that describe concurrency and nesting. State Charts are part of the *Unified Modelling Language* specification.

Sub-Class

A sub-class is a class that inherits data structures and methods from another class, which is called its super class. The sub-class may now define additional data structures and methods, or alter the methods of its super class by overwriting (i.e. re-defining) them. Depending on the programming language, a class may be a sub-class of one or multiple classes. Sub-classing is used to exploit *polymorphism*.

Synchronisation Algorithm

Synchronisation algorithms are used to ensure that the local causality constraint (see section 1.2.3) holds in PDES. Therefore, they *synchronise* the LPs with each other.

Unified Modelling Language (UML)

The Unified Modelling Language (UML) is a set of diagram languages to model and specify software systems from different points of view. For example, class diagrams are used to describe the class hierarchy of a software, whereas State Charts are used to specify the behaviour of a single object.

XML

The eXtensible Markup Language (XML) is a commonly used generic markup language to store arbitrary data. Its main advantages over proprietary file formats are its flexibility (i.e. there are mechanism to transform XML files) and that it is human readable. For more details, refer to the XML specification [92].

List of Figures

List of Tables

A Voting Scheme for Model Migration

Let u be the number of the (disjunct) sets of model entities, $V_1, \ldots, V_u$, which contain all model entities that have to be assigned to the p available LPs, $LP_1, \ldots, LP_p$. The number of model sets is bounded to the number of LPs, so $u \leq p$ holds. The model set holding the model that is managed by the topmost coordinator will be assigned to the processor that already hosts the topmost coordinator. The remaining $u-1$ model sets have to be assigned to $p-1$ LPs.

To do so, the voting scheme selects the $u-1$ maximal values $max_k = \max_{1 \leq i \leq u-k, 1 \leq j \leq p-k} M^k_{i,j}$ from non-negative $(u-k) \times (p-k)$ matrices M^k, $k = 1, \ldots, u-1$. Let M^0 be a $u \times p$ matrix. An element $m^0_{i,j} \in M^0$ is the number of model entities that are elements of model set V_i and currently hosted on LP_j.

M^1 is derived from M_0 by removing the row that corresponds to the model set of the topmost coordinator, and the column that corresponds to the LP which hosts it. Now, the voting scheme selects the maximal value of M^1, and again removes the corresponding row and column to generate the next matrix, M^2. This continues until the matrix is empty, i.e. there are no more elements left.

The amount of migrations induced by selecting a value $m^k_{i,j}$ is the sum of *all* other elements of the same row in M^0, i.e. the number of model entities of the same set that are currently hosted on other LPs. Hence, the migration number reduction for choosing $m^k_{x,j}$ instead of $m^k_{y,j}$ can be calculated by $m^k_{x,j} - m^k_{y,j}$. Let $o_1, \ldots, o_{u-1}$ be the optimal migration choice. It can be shown that the voting scheme does not result in more than max_1 unnecessary migrations:

$$\sum_{i=1}^{u-1} (o_i - max_i) \leq max_1 \tag{A.1}$$

Consider the case $u - 1 = 2$, so that inequation (A.1) can be written as $(o_1 - max_1) + (o_2 - max_2) \leq max_1$. The first summand, $o_1 - max_1$, is less or equal to zero, since max_1 is a maximal value of M^1. The second summand, $o_2 - max_2$, is positive if – and only if – o_2 is situated in the same row or column like max_1 [1]. The reason for this is the calculation of max_2, which is defined to be a maximal element of M^2.

Therefore, the left hand side of inequation (A.1) is maximal if (and only if) $o_1 = o_2 = max_1$ and $max_2 = 0$ hold. In this case, there are max_1 unnecessary migrations. The same reasoning can be applied to larger matrices. In those cases, the worst case will *still* induce an amount of unnecessary migrations that is less or equal to max_1. This can be proven by induction.

[1] In case $u - 1 = 2$, the second summand *has* to be greater or equal to zero. Otherwise, the optimal migration choice would *not* be optimal.

B UML Diagrams of SimSim

The following UML diagrams have been generated with Poseidon for UML (Community Edition 3.1-0) [32].

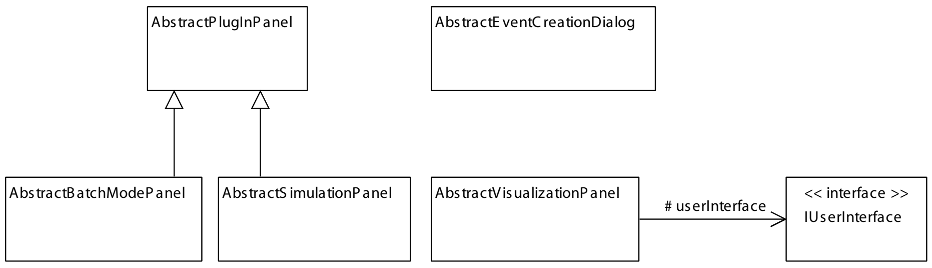


Figure B.1: UML diagram of package `gui.plugins`. This package contains classes that facilitate the development of plug-in user interfaces.

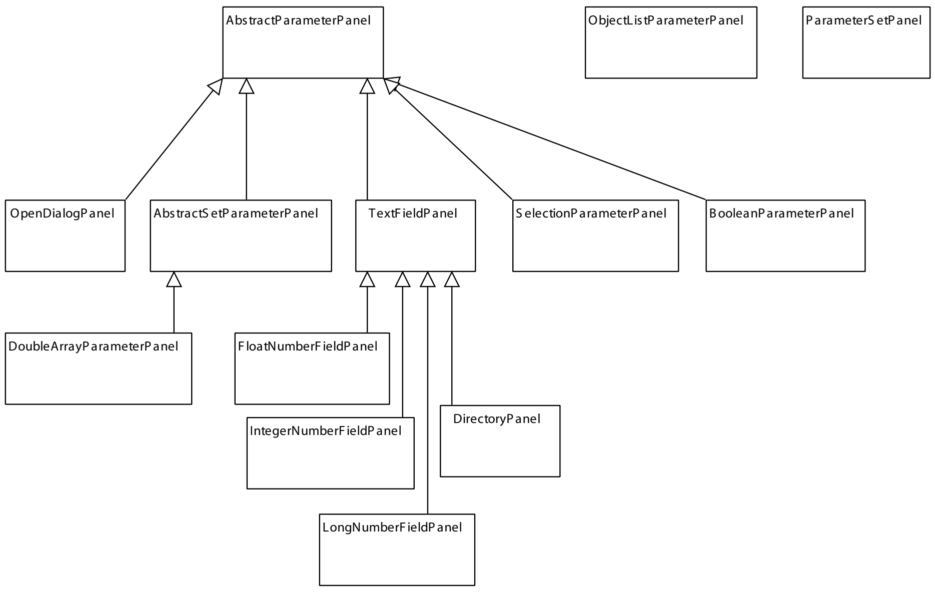


Figure B.2: UML diagram of package `gui.dialogs.components.parampanels`. This diagram supplements figure 4.7 by showing all parameter editing panels that are available.

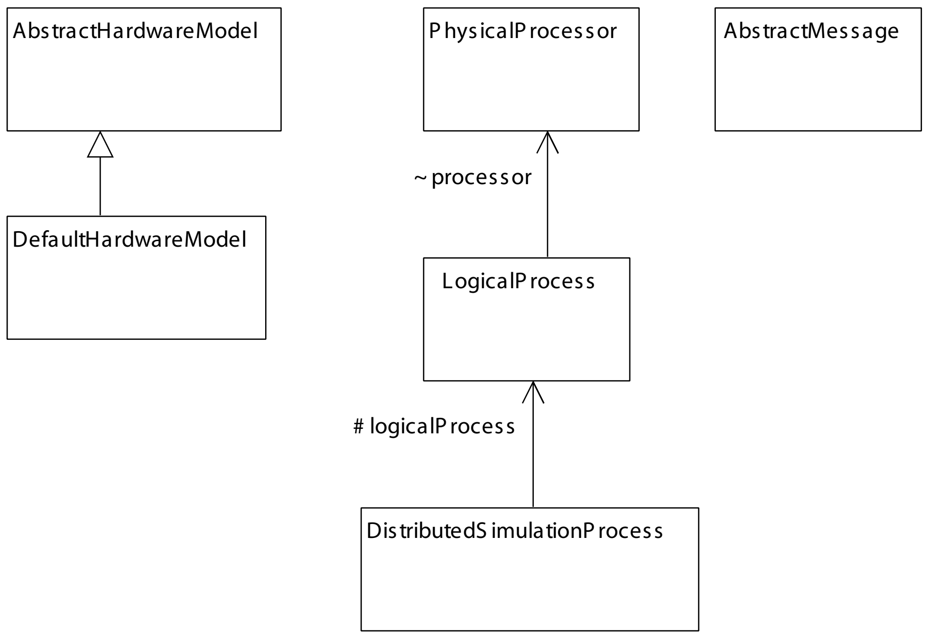


Figure B.3: UML diagram of package `model`. The `model` package contains the classes that form the generic PDES system model (see section 4.4).

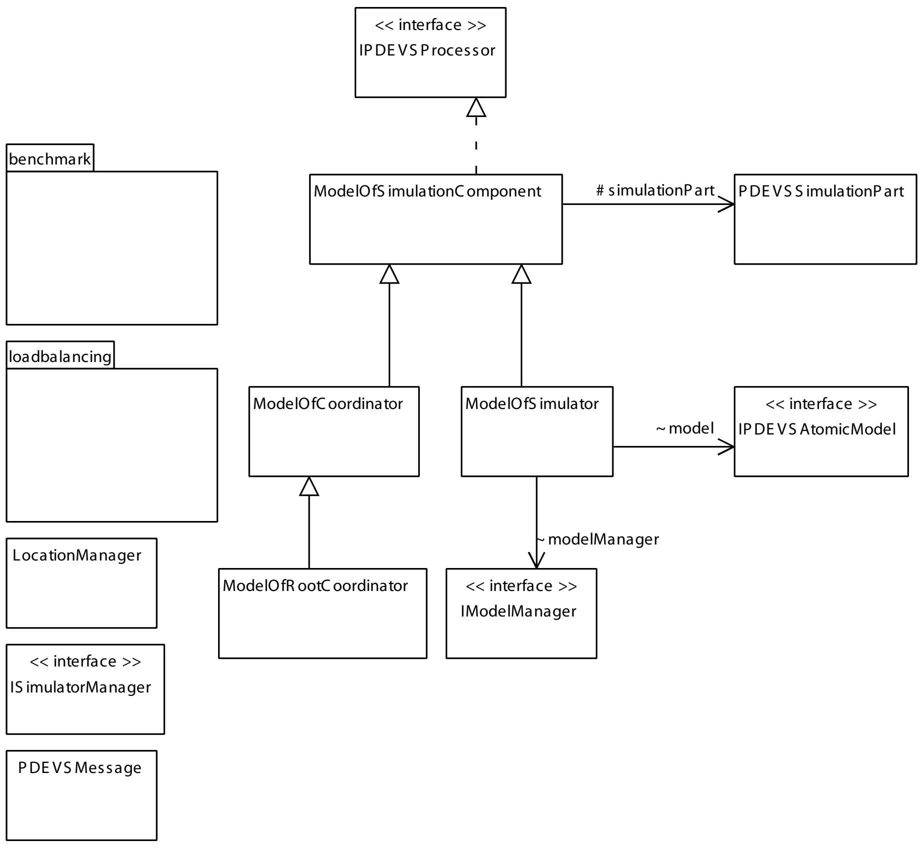
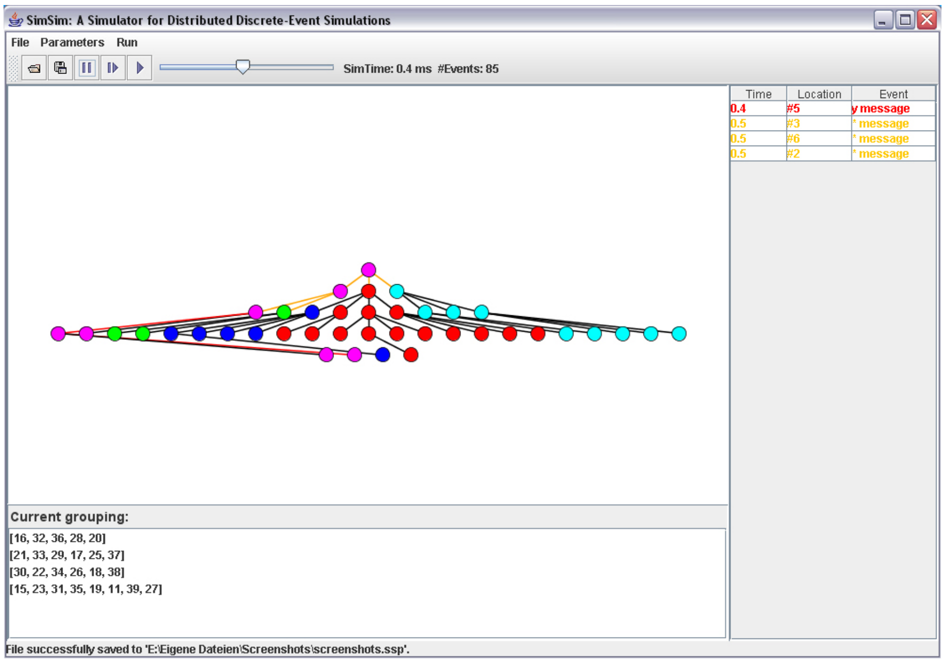


Figure B.4: UML diagram of package `plugin.james2.model`. Here, a part of the specific James II model is outlined. The corresponding application model can be found in sub-package `benchmark`, and the classes that implement the load balancing algorithm are separately kept in the `loadbalancing` sub-package.

C SimSim Screenshots


SimSim: A Simulator for Distributed Discrete-Event Simulations
File
Parameters
Run

SimTime: 0.4 ms #Events: 85

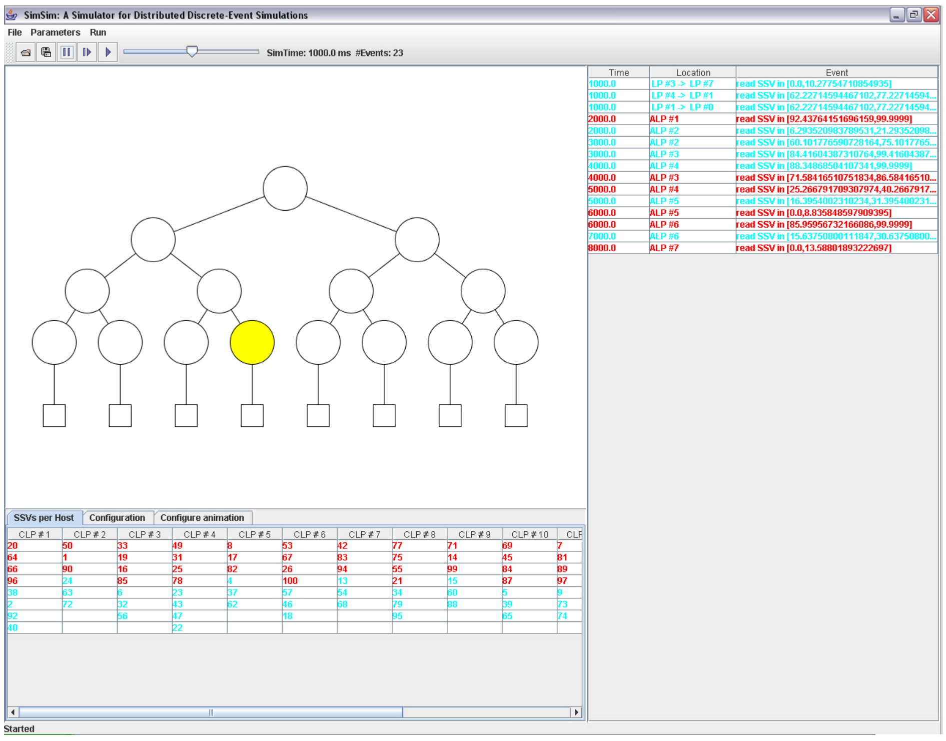
Time	Location	Event
0.4	#5	y message
0.5	#3	* message
0.5	#6	* message
0.5	#2	* message

Current grouping:

```
[16, 32, 36, 28, 20]
[21, 33, 29, 17, 25, 37]
[30, 22, 34, 26, 18, 38]
[15, 23, 31, 35, 19, 11, 39, 27]
```

File successfully saved to 'E:\Eigene Dateien\Screenshots\screenshots.ssp'.

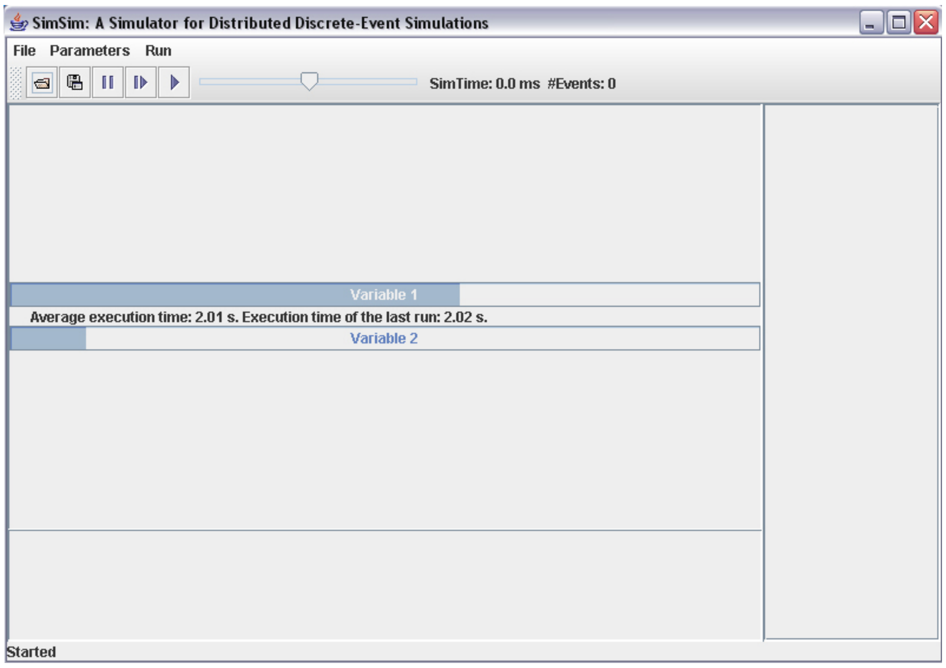
Figure C.1: Interactive Mode of SimSim: James II. This screenshot shows the interactive simulation of a PDEVS model. Above the rudimentary visualisation of the PDEVS model tree, several controls allow to stop and start the simulation, or to simulate step-wise (i.e. event per event). Moreover, the slider allows the adjustment of the simulation speed. The event table on the right shows the content of the event queue. To facilitate differentiation between PDEVS message types, each type has a certain colour. This colour is also used to animate the message transfers in the model tree. Below the visualisation, the current grouping of the atomic benchmark models is displayed, so this is a view on the application model.


SimSim: A Simulator for Distributed Discrete-Event Simulations
File
Parameters
Run
SimTime: 1000.0 ms #Events: 23
SSVs per Host
Configuration
Configure animation
Started

Time	Location	Event
1000.0	LP #3 -> LP #7	read SSV in [0.0,10.27754710854935]
1000.0	LP #4 -> LP #1	read SSV in [62.22714594467102,77.22714594...
1000.0	LP #1 -> LP #0	read SSV in [62.22714594467102,77.22714594...
2000.0	ALP #1	read SSV in [92.43764151696159,99.9999]
2000.0	ALP #2	read SSV in [6.293520983789531,21.29352098...
3000.0	ALP #2	read SSV in [60.101776590728164,75.1017765...
3000.0	ALP #3	read SSV in [84.41604387310764,99.41604387...
4000.0	ALP #4	read SSV in [88.34868504107341,99.9999]
4000.0	ALP #3	read SSV in [71.58416510751834,86.58416510...
5000.0	ALP #4	read SSV in [25.266791709307974,40.2667917...
5000.0	ALP #5	read SSV in [16.3954002310234,31.395400231...
6000.0	ALP #5	read SSV in [0.0,8.835848597909395]
6000.0	ALP #6	read SSV in [85.95956732166086,99.9999]
7000.0	ALP #6	read SSV in [15.63750800111847,30.63750800...
8000.0	ALP #7	read SSV in [0.0,13.58801893222697]

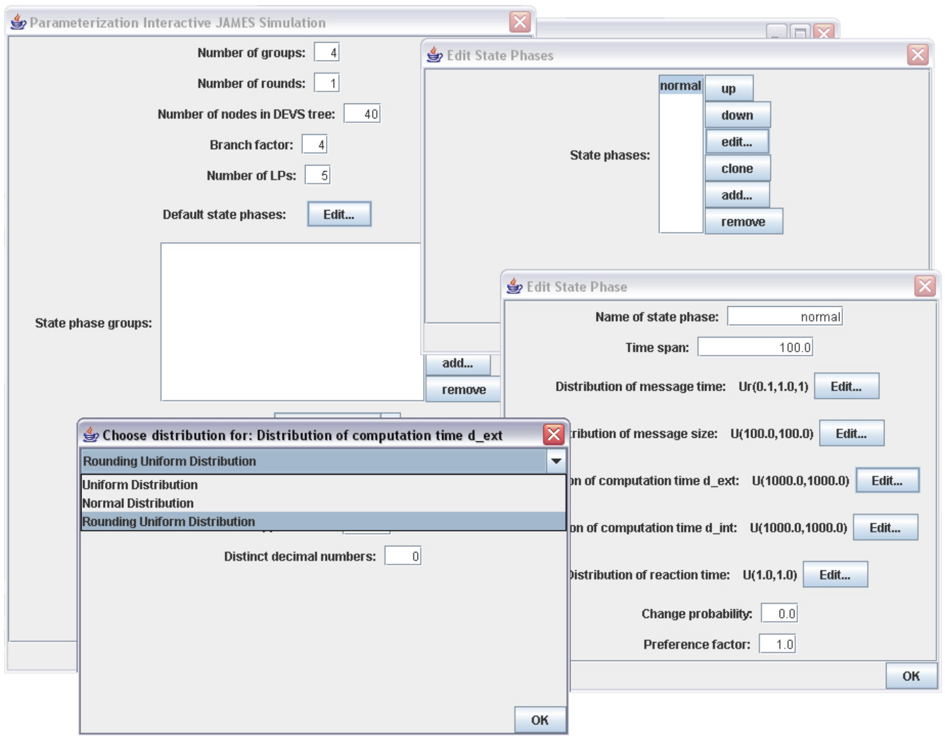
CLP # 1	CLP # 2	CLP # 3	CLP # 4	CLP # 5	CLP # 6	CLP # 7	CLP # 8	CLP # 9	CLP # 10	CL
20	50	33	49	8	53	42	77	71	69	7
64	1	19	31	17	67	83	75	14	45	81
66	90	16	25	82	26	94	55	99	84	89
96	24	85	78	4	100	13	21	15	87	97
38	63	6	23	37	57	54	34	60	5	9
2	72	32	43	62	46	68	79	88	39	73
92		56	47		18		95		65	74
40			22							

Figure C.2: Interactive Mode of SimSim: PDES-MAS. The visualisation is an incomplete port from the PDES-MAS log file visualiser. Here, the event queue contains different queries, which are coloured by the SSV type they concern. The table below the visualisation shows the current positions of all SSVs.


SimSim: A Simulator for Distributed Discrete-Event Simulations
File
Parameters
Run
SimTime: 0.0 ms #Events: 0
Variable 1
Variable 2
Started

Average execution time: 2.01 s. Execution time of the last run: 2.02 s.

Figure C.3: Batch Mode of SimSim. In batch mode, no visualisation is activated, since this could hamper the performance. Two progress bars in the centre show the progress of the batch job.



Parameterization Interactive JAMES Simulation

Number of groups: 4

Number of rounds: 1

Number of nodes in DEVS tree: 40

Branch factor: 4

Number of LPs: 5

Default state phases: Edit...

State phase groups:

Edit State Phases

State phases: normal

up
down
edit...
clone
add...
remove

Edit State Phase

Name of state phase: normal

Time span: 100.0

Distribution of message time: Ur(0.1,1.0,1) Edit...

ribution of message size: U(100.0,100.0) Edit...

on of computation time d_ext: U(1000.0,1000.0) Edit...

on of computation time d_int: U(1000.0,1000.0) Edit...

Distribution of reaction time: U(1.0,1.0) Edit...

Change probability: 0.0

Preference factor: 1.0

OK

Choose distribution for: Distribution of computation time d_ext

Rounding Uniform Distribution

Uniform Distribution
Normal Distribution
Rounding Uniform Distribution

Distinct decimal numbers: 0

OK

Figure C.4: Parameter Dialogues. As described in section 4.6, the parameter framework can be used to generate complex dialogues that manage parameters (in this case, the parameters of the benchmark model described in section 5.3).

Declaration

I declare that I conducted the work at hand independently, and aided by given literature and resources only.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Location / Ort, Date / Datum Signature / Unterschrift

Theses

Chapter 2

- The development of parallel and distributed discrete-event simulation (PDES) systems is a complex endeavour.
- Today's way of developing and evaluating PDES systems and algorithms lacks comparability, reproducibility, and falsifiability. Obtained results are often very specific and not broadly applicable.

Chapter 3

- The complexity of developing PDES systems is caused by the complexity of the models to be simulated.
- Mathematical approaches are useful for discovering theoretical bounds, but strongly limited when it comes to analysis of complex simulation systems.
- Since analytical simulations are typically used to analyse complex systems, simulation techniques can be used to analyse PDES systems.

Chapter 4

- A general purpose simulation simulator has to support the simulation of different PDES systems, and therefore should be extensible regarding parameterisation, measurements, and algorithms.
- Any PDES system can be regarded as a set of logical processors exchanging messages. This is the common basis of all PDES systems.

Chapter 5

- The gain of executing two PDEVS models on distinct hosts depends on communication cost, calculation cost, the extent of model activity, and the degree of parallelism between both models.
- Considering migration costs and applying a basic voting scheme to obtain new model placements reduces the number of migrations caused by the load balancing algorithm.
- Aided by SimSim, it is possible to detect design flaws of an algorithm at an early stage.

Chapter 6

- The use of SimSim allows to scrutinise an algorithm's behaviour under a variety of circumstances.
- Except for some specific circumstances, SimSim's James II - plug-in is able to predict the performance of James II's abstract PDEVS simulator with an acceptable accuracy (approximately 10% deviation from actual performance).

Chapter 7

- The approach of simulating PDES systems does indeed facilitate their development.
- Prior simulation of PDES systems can be integrated with software prototyping.