Automatic Algorithm Selection for Complex Simulation Problems

Dissertation

zur
Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
der Fakultät für Informatik und Elektrotechnik
der Universität Rostock

urn:nbn:de:gbv:28-diss2011-0162-1

vorgelegt von

Roland Ewald

Rostock, 13. 08. 2010

Verteidigung: 15. Oktober 2010

Gutachter:
Prof. Dr. Adelinde M. Uhrmacher (Universität Rostock, Germany)
Prof. Dr. David M. Nicol (University of Illinois at Urbana-Champaign, USA)
Dr. Georgios K. Theodoropoulos (University of Birmingham, UK)

Abstract

To select the most suitable simulation algorithm for a given task is often difficult. This is due to intricate interactions between model features, implementation details, and runtime environment, which may strongly affect the overall performance. An automated selection of simulation algorithms supports users in setting up simulation experiments, without demanding expert knowledge on simulation.

The first part of the thesis surveys existing approaches to solve the algorithm selection problem and discusses techniques to analyze simulation algorithm performance. A unified categorization of existing algorithm selection techniques is worked out, as these stem from various research domains (e.g., finance, artificial intelligence).

The second part introduces a software framework for automatic simulation algorithm selection and describes its constituents, as well as their integration into the modeling and simulation framework JAMES II. The implemented selection mechanisms are able to cope with three situations: a) no prior knowledge is available, b) the impact of problem features on performance is unknown, and c) a relationship between problem features and algorithm performance can be established empirically.

An experimental evaluation of the developed methods concludes the thesis. It is shown that an automated algorithm selection may significantly increase the overall performance of a simulation system. Some of the presented mechanisms also support the research on simulation methods, as they facilitate their development and evaluation.

Zusammenfassung

Die Auswahl des passendsten Simulationsalgorithmus für eine bestimmte Aufgabe ist oftmals schwierig. Dies liegt an der komplexen Interaktion zwischen Modelleigenschaften, Implementierungsdetails und der Laufzeitumgebung, welche die Gesamtleistung stark beeinflussen kann. Eine automatisierte Auswahl von Simulationsalgorithmen unterstützt die Anwender eines Simulationssystems, indem sie eine geeignete Konfiguration des Systems ohne zusätzliches Fachwissen aus dem Bereich Simulation ermöglicht.

Der erste Teil der Arbeit befasst sich eingehend mit Vorarbeiten zur automatischen Algorithmenauswahl, sowie mit der Leistungsanalyse von Simulationsalgorithmen. Eine einheitliche Kategorisierung wird erarbeitet um die aus den verschiedensten Fachgebieten (z.B. Finanzwesen, Künstliche Intelligenz) stammenden Ansätze zur Algorithmenauswahl einzuordnen.

Der zweite Teil der Arbeit stellt ein Rahmenwerk zur automatischen Auswahl von Simulationsalgorithmen vor und beschreibt dessen Bestandteile, einschließlich ihrer Integration ins Modellierungs- und Simulationsrahmenwerk JAMES II. Die realisierten Auswahlmechanismen beziehen sich dabei auf drei Situationen: a) keinerlei Vorwissen ist vorhanden, b) der Einfluss von Problemeigenschaften auf die Leistung ist unbekannt, und c) ein Zusammenhang zwischen Problemeigenschaften und Algorithmenleistung konnte empirisch ermittelt werden.

Eine experimentelle Untersuchung der entwickelten Methoden schließt die Arbeit ab. Es wird gezeigt, dass eine automatische Algorithmenauswahl die Gesamtleistung eines Simulationssystems signifikant steigern kann. Einige der vorgestellten Mechanismen unterstützen außerdem die wissenschaftliche Untersuchung von neuen Simulationsmethoden indem sie deren Entwicklung und Evaluierung vereinfachen.

CR–Classification (1998):
I.2.6[Artificial Intelligence]:Learning — Induction, Knowledge Acquisition;
I.6.0[Simulation and Modeling]:General — Performance;
I.6.7[Simulation and Modeling]:Simulation Support Systems — Environments;
I.6.8[Simulation and Modeling]:Types of Simulation — Discrete event, Parallel;

Key words: Simulation Algorithm Selection, Algorithm Selection Problem, Adaptive Replication, Simulation Algorithm Portfolios, JAMES II

Contents

1. Introduction

Computers are useless. They can only give you answers.

Pablo Picasso

1.1. Motivation

Modeling & Simulation

How can computers help us? This question, at the very foundation of (applied) computer science, is not yet answered conclusively. There are many fields where computer systems are widely accepted, e.g., administration or education, even though their positive impact is debatable [33, 261, 326]. If there is any aspect of human activity for which the surplus of using computers is *not* questionable, this is the scientific endeavor. Computers have profoundly changed and enhanced research methodology: petabytes of measurement data are stored and processed to interpret experiments with the Large Hadron Collider [285], efficient algorithms help analyzing DNA sequences [2], and whole experimentation processes are automated by computers and robots [178]. Apart from merely processing symbolic data faster and less error-prone than humans, computers are also valuable tools for testing our *hypotheses* regarding a system under investigation. This can be done by abstracting away all aspects of the system that are not essential to testing the hypothesis, thereby forming a formal, abstract specification of the system: a computer *model*. Computer models may be analyzed in various ways, e.g., by calculating the model's behavior via computer *simulation* (more precise definitions are given in sec. 1.2). The given hypothesis could now be falsified by the simulation outcome. From this perspective, the field of *modeling and simulation* is concerned with the computerized support of the *scientific method* in general, ideally serving as a catalyst for scientific progress (see sec. 1.4).

Simulation as a Scientific Discipline

Modeling and simulation is a scientific discipline in itself, concerned with providing valid and efficient techniques for model construction and execution. Simulation relies on many sub-domains of computer science. Basically, it can be divided into three aspects: the conceptual development of simulation algorithms, their efficient and valid implementation, and the creation of environments in which these implementations can be used productively by others, not necessarily computer scientists. For example, there are many methods for numerical integration, a technique that is used for simulating continuous systems [34, p. 57 et sqq.]. Besides their mathematical properties, which can be enhanced by conceptual developments, their efficient implementation on modern computers poses several additional challenges (e.g., due to rounding [34, p. 28 et sqq.]). Eventually, these efficient implementations have to be made available to non-expert users, e.g., via simulation environments like Simulink [224].

Devising simulation algorithms usually incorporates ideas from numerical and discrete mathematics, algorithmic complexity theory, and parallel algorithms. Implementing efficient simulation algorithms requires knowledge on software engineering, programming languages, operating systems, computer architectures, and networks. Environments for using the implemented algorithms often have graphical user interfaces and rely on many other fields of applied computer science, particularly databases and visualization.

From a scientific point of view, a proposed simulation algorithm comes along with the hypothesis that it works *correctly*, i.e., it yields valid results, and *efficiently*, i.e., it does so with parsimonious use of resources like memory or CPU time. A *new* simulation algorithm should also compare favorably to

existing approaches in these regards, to justify its usage. Implementing and running the algorithm can be regarded as an experiment to test this hypothesis [142, p. 203]. The scientific method demands that a hypothesis can be *falsified*, i.e., it must be possible to disprove the hypothesis on factual grounds. For example, the hypothesis associated with a newly proposed simulation algorithm can be falsified by showing that it is less accurate and efficient than an algorithm that is already known — however, the unbiased comparison of simulation algorithms is not easy. Algorithmic complexity theory allows to theoretically compare algorithms regarding their resource consumption. Unfortunately, there are many pitfalls and limitations when these results shall be transferred to reality (see sec. 2.2).

Empirical comparison of algorithms is not easy either. Implementations are given in various programming languages. Compilers, operating systems, hardware components, and network devices usually interfere strongly with the efficiency of the implementation. Furthermore, benchmarking is not standardized, so often poorly documented benchmark models are used to 'proof' a point (see ch. 7). This hampers falsification and therefore also the progress of the field itself: How can I know that I will achieve roughly the same speed-up as stated in the article, when the authors implemented the algorithm in language X on machine Y to simulate model Z, whereas I need to simulate model A on machine B and would prefer language C? Apart from some general insights into common bottlenecks, the simulation community leaves practitioners rather helpless when they have to choose a suitable simulation algorithm for a given problem.

This situation casts doubt on computer simulation as a science: If it is not possible to falsify the hypothesis associated with a new simulation algorithm, i.e., it is essentially incomparable to existing methods, *how can its scientific contribution be evaluated?* Although the simulation community is aware of this issue [291], it seems as if there is no simple solution. The stakes are high: complaints about lacking scientific rigor in computer science are anything but new [59], but simulation risks to become a *pathological science* [332] if these problems are not solved, i.e., a science that lacks reproducibility and therefore fails to yield valuable insights. Automating the overall research process seems to be an interesting possibility to overcome some of these difficulties [320].

Towards Simulation Algorithm Comparability: Modeling & Simulation Environments

The aforementioned problem can be partly solved by standardization: if researchers rely on the same algorithm environment and programming language, they facilitate a fair evaluation of their contributions. This is one of the motivations that drive the development of general-purpose modeling and simulation frameworks. Such frameworks allow a fair comparison in principle, but *general* statements about an algorithm's performance are still hard to come by: most simulation algorithms strongly depend on model characteristics, and the underlying hardware and operating system might still bias the results. This leaves us in a situation where it is easy to compare two simulation algorithms for *one* specific model, executed on *one* specific computer — but it is still unclear how this relates to *another* situation. General-purpose modeling and simulation environments make simulation algorithm comparisons reproducible and fair, but using them does not suffice to get the big picture. How can we know which algorithms are indeed promising for a broad class of models, and which only work on a small set of carefully crafted benchmarks? Which model properties influence the performance of an algorithm the most? When is algorithm A preferable over algorithm B?

A Case for Simulation Algorithm Selection

Without general, comparative insights into existing algorithms, it is almost impossible to take the right design decisions when devising and implementing new ones — only intuition can guide the developer, often accompanied by trial and error. A similar problem arises on the side of the user: when a simulation system provides many ways to solve the problem at hand, which one is the most appropriate? Are all of them valid and stable? At the core of both problems is the so-called *Algorithm Selection Problem*, i.e., the problem of selecting a suitable algorithm for processing a given input with given resources (hardware, operating system, etc.). Solving this problem may allow simulation systems to configure themselves *automatically*. This will help users, and also implies obtaining *general* knowledge

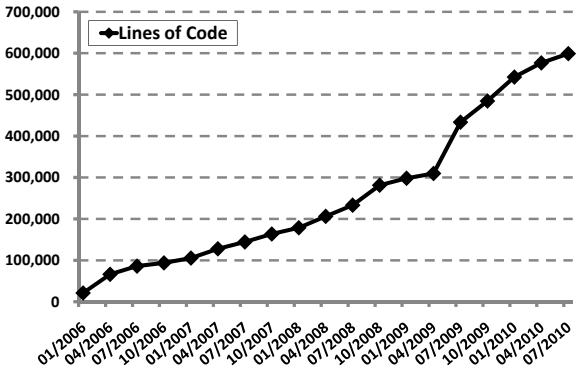

Figure 1.1.: Growth of the JAMES II code base over the past years.

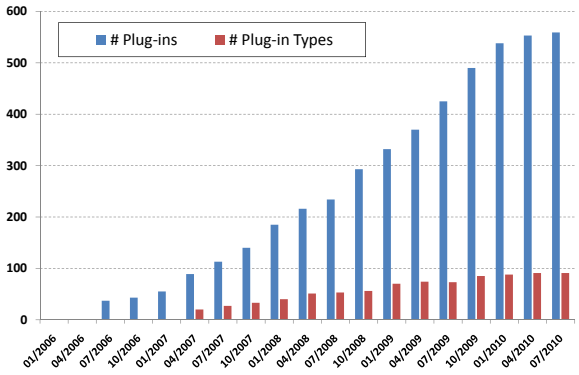

Figure 1.2.: Number of plug-ins and plug-in types defined in JAMES II (see sec. 4.2, p. 81).

on the relative merits of simulation algorithms. Researchers will benefit from such knowledge as well, as it may reveal the weaknesses of current methods and could point to promising research directions.

This thesis addresses the Algorithm Selection Problem in a simulation context. The presented methods are implemented within the modeling and simulation framework JAMES II, which was constructed to serve as a productive environment for comparing simulation algorithms [131] and has been growing steadily over the past years (see fig. 1.1 and 1.2). The abundance of competing algorithmic solutions for certain simulation tasks in JAMES II is one of the more practical motivations to investigate their automated selection. Nevertheless, the developed methods are not restricted to JAMES II; they are applicable to simulation systems in general.

The next section defines some important terms and concepts. Section 1.3 introduces two real-world challenges for simulation algorithm selection, which serve as example domains in the following. Finally, section 1.4 briefly overviews the philosophical grounds on which the objectives of this thesis can be evaluated, while section 1.5 concludes the introduction by outlining the structure of the next chapters.

1.2. Terminology

The notion of an *algorithm* is central to computer science. The most prominent definition is given by Turing, who proposed a *Turing Machine (TM)* in analogy to a person that is computing a number [307, p. 231–233]. The person only has a finite memory, but may use (infinitely many) segments of a tape to keep notes. One can now think of a machine that shall *automate* this process: it has a finite number of states (representing its memory), it is able to read from or write to the tape (one symbol at a time), and it may also move left or right along the tape (segment-by-segment). Turing regards a machine as *automatic* if its next action is always completely determined by its current position on the tape, its state, and the content of the tape. This notion is contrasted by the definition of a 'choice-machine'; a machine that sometimes requires intervention by an 'external operator', i.e., which relies on elements that are not necessarily computable.

However, the *automatic* version of Turing's machine model does not rely on such interventions. It is this machine model that can be used to define what an algorithm is: everything that can be described by an (automatic) Turing machine. Algorithms might be given in verbal form or as code in some programming language; they all have in common that they can be mimicked by a Turing Machine. Nevertheless, the argument that this kind of machine does indeed cover *all* possible processes of calculation remains unproven: *"All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically"* [307, p. 249]. Similar to Turing, Davis et al. conclude that *"[...] the word algorithm has no general definition separated from a particular language [...]"* [54, p. 69], and the languages they refer to are programming languages

with formally defined semantics. The claim that Turing machines (and their equivalents) indeed characterize all functions that are 'effectively calculable' is referred to as *Church-Turing-Thesis* [315] and is widely accepted. It can also be generalized and grounded on general physical laws [58]. Apart from the notion of algorithms, Turing defined the term *automatic* as the absence of external (human) intervention. Consequently, the automatic selection of algorithms means that one out of several algorithms is chosen without human intervention. A formal definition of the problem is given in section 2.1.

The second part of the title restricts the topic to *complex simulation problems*. The definition of simulation depends on the notions of *system* and *model*. The term 'system' originates from the Greek *sustēma*, which combines *sun* (together) and *histanai* (to set up): things that are put together, i.e., *"a group of interacting elements"* [258, p. 832]. There are many meanings of 'model' in everyday life; in a simulation context, models are regarded as descriptions of systems [24, p. 1]. *Simulation models*, i.e., models that can be simulated by a computer, usually take the form of *"mathematical, logical, and symbolic relationships between the entities"* [14, p. 3], i.e., their semantics are *formally* defined.[1]

Simulation is the process of *"driving the model with certain [...] inputs and observing the corresponding outputs"* [24, p. 2], i.e., it can be regarded as an experiment on a given model [34, p. 8]. This thesis deals with (digital) computer simulation, i.e., the simulation of formal models by an algorithm that runs on a digital computer — in contrast to purely analog simulation, e.g., of an airplane model in a wind tunnel, or simulation by an analog computer. A simulation *imitates* the modeled system to some extent [14, p. 3], resembling the popular meaning of 'simulation'. What makes simulation problems complex is more formally discussed in section 2.4. For now, a simulation problem is deemed *complex* if it is difficult to decide beforehand which simulation algorithm will be the most appropriate to solve it.

1.3. Examples

This section presents two domains of simulation research that could benefit from an automatic simulation algorithm selection. The first domain is application-specific and belongs to computational *systems biology* [180, 242], an emerging field that aims at further integrating methodology for systems analysis, e.g., modeling and simulation, into biological research. The second domain, parallel and distributed discrete-event simulation (PDES), is a fundamental area of simulation research and is rooted in research on distributed computer systems [36, 233]. Its approaches are application-independent and can thus be transferred to various domains, including the problems from systems biology presented in the first example. Due to the advent of multi-core CPUs in ordinary personal computers [130], the field still holds many promises for speeding up simulation execution [92].

Both domains have in common that a bevy of simulation methods and enhancements exists, most of which highly depend on the concrete problem at hand. Many are poorly evaluated and not thoroughly compared to competing techniques. At the same time, ever more complex simulation systems are demanded to guarantee a valid, fast, and stable execution [247]. This leads to a '*simulation gap*' between the development and maintenance of large, general-purpose simulation systems on the one hand, and the development of highly application-specific simulation algorithms on the other hand: a general-purpose system cannot incorporate highly tuned algorithms for special cases without providing a convenient algorithm selection mechanism. Solutions for the algorithm selection problem in both example domains, realized by the methods introduced in this thesis, are evaluated in chapters 9 and 10.

1.3.1. Simulation of Chemical Reaction Networks

Systems biology employs various modeling and simulation methods for analyzing the organizational levels of an organism, ranging from molecular dynamics on the level of atoms [314], over the simulation

[1] On a more abstract level, models can also be understood as composite hypotheses regarding a system, i.e., a hypothesis that incorporates parameters to be set [84, p. 408].

of chemical reaction networks, to the simulation of whole cells, organs, and organisms [312]. Each level of abstraction poses several challenges to modelers and simulation developers alike, and can be specified by various modeling formalisms [73]. An abstraction level that is widely popular among systems biologists is that of chemical reactions, i.e., considering dynamics on the level of molecules. It can be used to describe many cell-biological phenomena, such as protein-DNA interactions [56], metabolic reactions [156], or signaling pathways [197].

There are several ways to describe a system on the level of molecules. Individual-based approaches, such as biologically inspired extensions of the DEVS [340] formalism [309, 310], have some advantages. They avoid a state-space explosion that occurs when considering molecules that could be modeled with the notion of a state, e.g., proteins with several phosphorylation sites [306]. Furthermore, some of them allow to take additional aspects into account, e.g., the spatial distribution of reactants [162] and the size of molecules [161]. This allows to model effects like *molecular crowding*, i.e., the clustering of certain molecules, which may greatly impact system behavior [179]. Likewise, simulation algorithms are adapted to the new requirements, e.g., to run on multiple resources or to support spatial models (e.g., [159, 161]). However, most of these approaches are based on the so-called Gillespie algorithm [105, 106].

Gillespie argued that the continuous simulation approaches for chemical reaction networks, which are based on ordinary differential equations (ODEs), do not allow to assess the true behavior of the system if small quantities of particles are involved — which is the case in many biological systems, e.g., when considering viral kinetics [295]. Instead, he reformulated the problem as a *continuous-time Markov chain (CTMC)*, i.e., as a system undergoing random discrete state transitions on a continuous time scale, with probabilities that only rely on the current state [106]. Although these can be analyzed numerically in certain cases (e.g., see [28]), larger models usually require a stochastic simulation of the possible trajectories along which the model may evolve. This requires to repeat a simulation multiple times with different random numbers, to get statistically sound results. The repetitions are also called *replications*. They are necessary to obtain general insights into the behavior of stochastic models.

Gillespie introduces two *stochastic simulation algorithms* (SSAs), the First Reaction Method and the Direct Method. Both are equivalent in terms of the results they produce, but not with respect to their runtime performance (see ch. 9). Other SSAs that incorporate advanced data structures and new algorithmic ideas have been proposed over the years, e.g., [31, 104], although their merits are still up for debate [281]. Additionally, approximation techniques like *τ-leaping* have been introduced to speed up execution [108]. They approximate a solution by executing more than one chemical reaction per step, thereby abstracting away changes of the reaction propensities in-between these steps. This can speed up simulation by several orders of magnitude. On the other hand, such approximation might not be accurate enough to answer certain questions about the system. For example, the τ-leaping approach from [30] may introduce a statistically significant error when observing the number of particles over time [158]. It is therefore important to know when to use the approximation, and when to better stick to the 'exact' SSA variants.

Accounting for both aspects of SSA simulator performance, i.e., execution time and accuracy, is still a task that has to be done manually; which algorithm performs best in which situation has not been settled yet. Nevertheless, SSAs are widely used today: some approaches have been built on top of them, e.g., the Next Subvolume Method for simulating diffusion processes in a discretized space [62]. There are also other simulation algorithms that rely on SSAs, e.g., for the stochastic π-calculus [264]. Finding out which SSA variant is best for which model will therefore allow the optimization of several related methods.

Even with the most sophisticated tools and methods, systems biology remains an extremely challenging field where researches are still just beginning to get some insights [32] — but today's simulation tools even struggle with comparatively small models. One way to enhance the capabilities of current simulation systems is to support parallel and distributed simulation. This would enable researchers to simulate larger models and may also speed up simulation execution. For example, the spatial structures introduced by the Next Subvolume Method can be exploited by a parallel simulation scheme on top of SSAs [159, 160]. Bringing complex algorithms with not yet thoroughly investigated perfor-

mance characteristics to parallel computing is a daunting challenge — even more so, as the Algorithm Selection Problem manifests itself in the field of parallel and distributed simulation as well.

1.3.2. Parallel and Distributed Discrete-Event Simulation

The need for parallel and distributed simulation arises with very large and computationally challenging models that can only be simulated conveniently by combining existing resources. The parallel simulation of *discrete-event* models goes back to the late 1970s [91]. Discrete-event models have a continuous time basis, but are restricted to a finite number of state transitions per time interval [14, p. 12]. They are used in many important application areas. For example, the SSA variants introduced in section 1.3.1 compute a discrete-event model of a chemical reaction network, and hence are discrete-event simulators. Other domains of discrete-event simulation are computer networks [241] or manufacturing [14, 251].

Parallel and distributed discrete-event simulation is also applied to analyze *multi-agent systems (MAS)* [304], which are capable of adaptive behavior and can be used to implement automatic algorithm selection, as discussed in section 2.4. A multi-agent system consists of several *agents*, i.e., potentially complex entities that act (to some extent) autonomously [279, p. 4] within a certain environment. Agents may communicate with each other, either directly or via the environment, e.g., to solve some task. Modeling and simulation of MAS allows to assess the performance of artificial MAS under controlled circumstances, before an expensive and potentially dangerous prototype has to be implemented, e.g., for large-scale automated vehicle routing [328]. Furthermore, MAS simulation is used to investigate natural systems that suit this metaphor, e.g., everyday traffic phenomena [297] or crowd behavior [13] (see sec. 2.4.2).

Basic Execution Scheme

A *parallel simulation* allows to compute certain aspects of the model behavior in parallel, while a *distributed simulation* makes use of spatially distributed resources, i.e., computers connected over some kind of network. It is possible to conduct a parallel simulation that is not distributed, e.g., by employing components for parallel computing on a single machine, such as GPUs [253, 280, 297]. Likewise, it is also possible to run a sequential simulation in a distributed manner — but usually, discrete-event models are distributed over several computers to *allow* a parallel computation, hence the term *parallel and distributed discrete-event simulation (PDES)* [90].

Stemming from research on distributed systems in general, PDES algorithms are usually defined on the concept of a *logical process (LP)* [233, 321]. Each model entity can be regarded as such a process, and the processes that belong to a model exchange time-stamped *events*, usually via messages, to interact with each other. This is easy to implement as a sequential simulation, where the events are stored in time-stamp order by a central *event queue*. The event with the smallest time stamp is dequeued and propagated to its destination LP, where its effect will be calculated and newly generated events (with later or equal time stamps) might be created. These are added to the queue and the algorithm starts over, until the queue is empty or a termination condition is met. Several complications arise when executing this algorithm in parallel, on a distributed set of machines. Most importantly, the logical processes now run in the absence of a central event queue, which makes it necessary that they synchronize themselves. A *synchronization* protocol for PDES ensures that the *local causality constraint* holds [90, p. 52], i.e., it preserves the order with which events are processed at each LP. Figure 1.3 illustrates this fundamental difference between sequential and parallel discrete-event simulation.

Synchronization

Synchronization protocols can be divided into two large groups: *conservative* and *optimistic* ones. With a conservative protocol, each LP blocks the execution of an event until it can be sure that no event with a smaller time stamp will arrive in the future. This may lead to a *deadlock*, i.e., a situation

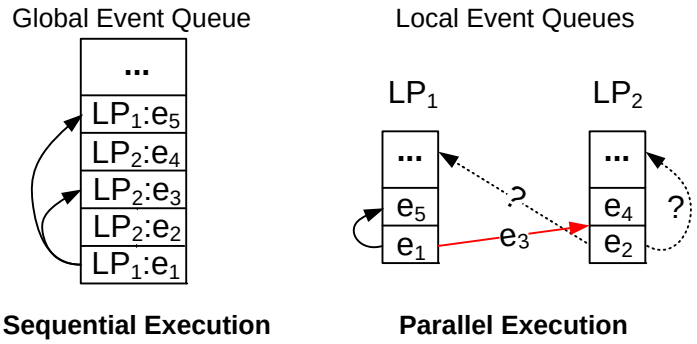

Figure 1.3.: Sequential vs. parallel discrete-event simulation: While the sequential processing of events merely requires a global event queue to ensure a correct computation order (left), the parallel execution has to *synchronize* the local event queues (right). Otherwise, if LP_1 processes event e_1 for too long, the newly generated event e_3 might not reach LP_2 before e_2 has already been processed and e_4 is executed instead. This violates the local causality constraint, which states that each LP has to execute its local events in the correct order. It ensures the equivalence to results from sequential simulation.

where no LP is sure that it can proceed and hence all LPs block each other, so that the simulation halts. Deadlocks can be avoided or detected by several methods [90, p. 60 et sqq.]. With an optimistic protocol, LPs just execute the event with the smallest time stamp in their local queue, optimistically assuming that there will be no event with a smaller time stamp on its way. If this assumption is wrong, the execution violates the local causality constraint. Hence, the arrival of an event with a time stamp smaller than the *local virtual time (LVT)*, i.e., the time stamp of the last event that has been executed locally, will cause a *roll-back* to a state where the local causality constraint has not been violated yet. Events that cause roll-backs are called *straggler events*. A straggler event induces a roll-back, which might in turn induce roll-backs at other LPs. These need to be notified in case they received invalid events from the given LP while it was violating the local causality constraint. Such cascades of roll-backs are a major problem of the classic optimistic protocol, *Time Warp* [157] (see sec. 10.1, p. 199), and again there are several techniques and protocol enhancements to alleviate it (e.g., lazy cancellation [90, p. 129–132]). The behavior of synchronization protocols is hard to evaluate and forecast. Entire simulation systems have been dedicated to their investigation, e.g., WARPED [221] was originally developed to be *"freely available to the research community for analysis of the Time Warp design space"* [220, p. 1].

Further Aspects: Partitioning, Load Balancing, Message Passing, Interest Management

Before executing a PDES with some synchronization protocol, model and simulator entities have to be distributed over the set of available machines. This is done by a so-called *partitioning* algorithm, which aims at assigning a fair share of computational load to each machine, and also at minimizing the inter-machine communication that occurs during the simulation. The latter is relevant because network latency slows down event propagation and may therefore hamper the overall execution speed. Network latency would be minimal if all entities were mapped to a single processor, but this would also result in a maximal imbalance, as only one processor would have to do all the work. On the other hand, perfectly balancing the load between all processors could lead to excessive network traffic. Partitioning has to find a good trade-off between both aspects. It is related to the *graph partitioning* problem from theoretical computer science, which makes a broad range of algorithms applicable to its solution [83].

Additional challenges arise in the context of model partitioning, where the main task is to find a mapping between the model graph and the graph that represents the (potentially heterogeneous) infrastructure [69]. Another problem is that a model's behavior is not fully known before runtime—

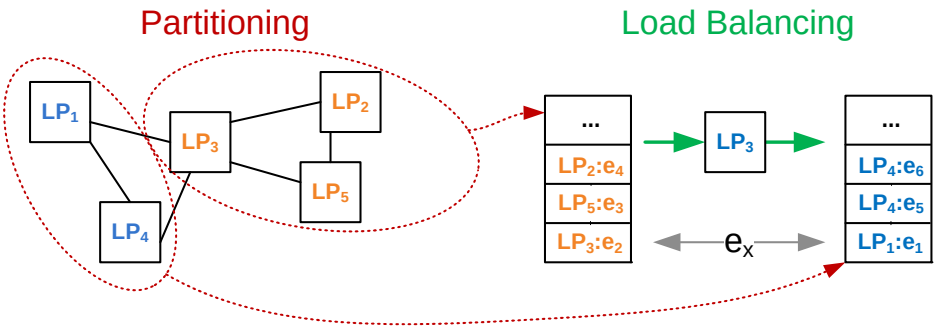

Figure 1.4.: Partitioning and load balancing: partitioning (red) occurs before the simulation starts. It aims at minimizing the communication of events between the local event queues that are executed in parallel (see fig. 1.3) and at fairly distributing the computational workload. Load balancing (green) is invoked at run-time to re-balance the workload if the initial partition turns out to be sub-optimal. In the given example, a load balancing algorithm might decide to relocate LP_3 if it turns to communicate much more with LP_1 and LP_4 than with LP_2 and LP_5.

otherwise, there would be no need to simulate it. This, in turn, implies that the original partitioning is not necessarily a good one. Even worse, the quality of the partitioning may deteriorate over time if the model exhibits certain kinds of dynamic behavior or structure, i.e., if the computational load changes during simulation. This problem is addressed by *load balancing* algorithms, which shall detect imbalance in a distributed simulation and then reduce it by migrating LPs from overloaded to underloaded machines.

Further algorithms are needed for tasks that are specific to the distributed simulation application at hand. The simulation of multi-agent systems, for example, might be enhanced by domain-specific synchronization algorithms [249] or by maintaining a shared state to which all LPs in the simulation have access [211]. Such novel approaches to distributed simulation may require adjusted schemes for load balancing [244], synchronization [198, 199], or even message routing [38, 66]. Most of these algorithms strongly rely on the characteristics of the multi-agent system at hand, and may also influence each other. For example, the performance of the routing algorithm may strongly depend on the algorithm that balances the load [38]. In other applications of distributed simulation, e.g., distributed multi-player games [219], related techniques like *interest management* are of central importance and need to be adjusted to the problem at hand [232]. Again, the specifics of the application are reflected in alternative algorithms for basic tasks such as synchronization (e.g., see [48]).

All in all, parallel and distributed discrete-event simulation is realized by a whole *ensemble* of algorithms that *interact* with each other, and also depend on the characteristics of the model and the hardware infrastructure. Moreover, specific simulation requirements, e.g., real-time capabilities for the emulation of computer networks [241], call for the development of ever more solutions. Since it is "*very difficult to analytically prove anything about performance executing large models on large machines*" [238, p. 4], empirical PDES performance prediction has been established over the past years (see sec. 3.3.2, p. 73). The *automatic* selection of PDES algorithms could be regarded as the next step toward user-friendly PDES tools, since it is hard to find a suitable combination of algorithms for the application at hand [70].

1.4. Epistemological Viewpoint

The main objective of this thesis, i.e., to develop methods for the automatic selection of suitable simulation algorithms, implicitly coincides with streamlining research on simulation algorithms itself. A successful selection mechanism requires *knowledge* on the performance merits of the available algorithms. Such knowledge is also useful in other contexts, such as the development of new simulation

algorithms — pursuing and obtaining it is an everyday task in simulation research.

From a philosophical perspective, this thesis is mainly concerned with *inductive prediction*. Induction is understood as inferring general laws from a sequence of observations. Here, observations relate algorithm implementations to performance measurements. Induction can be implemented by certain algorithms, which take the observations as input and return a hypothesis regarding the process that generated them. However, induction does not ensure that the laws derived from those observations are *true*. This is known as the *problem of inductive knowledge*, and has been pointed out by many philosophers of science, e.g., David Hume, Karl Popper, and Bertrand Russell [282, 300]. A classic anecdotal example for wrong induction is Russell's *Inductivist Chicken*: imagine a chicken that is always fed at 9 a.m. Being an inductivist, it collects observations from various weekdays and under various circumstances, e.g., hot and cold weather. Inevitably, it will conclude the general law *"I will always be fed at 9 a.m."* from the observations — but this law fails to predict that the chicken gets slaughtered the day before Christmas [300, p. 40].

The *hypothetico-deductive method* aims at circumventing this problem. It limits the role of induction to the confirmation of hypotheses *after* they have been formulated. Popper's falsification paradigm, i.e., to abandon confirmation in favor of theory falsification, goes along the same lines and can be interpreted as an *"anti-inductivist version of the hypothetico-deductive method"* [282, p. 253]. The method leaves theory formation to the scientist with domain knowledge, i.e., Popper distinguishes *"[...] sharply between the process of conceiving a new idea, and the methods and results of examining it logically"* [260]. This distinction makes the hypothetico-deductive method hard to automate, as it would require to incorporate some kind of artificial creativeness. Similarly, results from theoretical computer science suggest that a purely deductive automatic mechanism is infeasible for algorithm selection (see sec. 2.1.3).

Hence, automated induction remains — despite its pitfalls and shortcomings — the most suitable approach to be pursued here. Besides the *inductive skepticism* due to the problem of inductive knowledge, one might also ask how the performance of different inductive methods could ever be assessed, given that induction itself is fundamentally flawed from an epistemological perspective. Thomas Kuhn identified accuracy, consistency, simplicity, breadth, and fertility of a hypothesis as quality criteria [282]. On a more practical level, experimental soundness, control over sources of error and bias, and reproducibility are named as important methodological aspects of observing algorithm performance [163].

By taking all these requirements into account, one may take the view of an epistemological naturalist [282]: clearly, inductive methods are not sufficient to *prove* a hypothesis to be true — but maybe this degree of justification is simply unrealistic? While in principle *not* being able to identify the induction method that provides the best hypotheses, it is still possible to asses the quality of induction results by the aforementioned general criteria, and also by testing them in practice. In other words, the core argumentation line of this thesis retreats to a position of *scientific pragmatism*. The *pragmatic principle* finds it justified to accept a proposition P about the world if and only if one of the following conditions holds [5, p. 91]:

- P is inferred directly from other known or justified beliefs.
- It is likely that accepting P is more productive with respect to explanations and predictions than denying or ignoring P.

This definition is remarkable in that it is not related to the *truthfulness* of P, but rather to its *productivity* when it comes to *predictions*. In fact, *"[...] pragmatists generally agree that the truth or justification of a belief is less a function of how the belief originates than it is of whether the belief, however it originates, leads to successful predictions"*[5, p. 92]. Following the pragmatic principle, an inductively derived algorithm selection mechanism will be preferred over another if (and only if) its predictions are more accurate than those of the other. Likewise, an inductive method to generate selection mechanisms will be preferred over another if (and only if) its generated selection mechanisms are preferable to those of the other.

Radical pragmatists dismiss the notion of truth as not meaningful at all. *Non-radical pragmatists* do not deny the concept of truth, but maintain that although we cannot proof any of our beliefs about the real world, some of them might actually be true (at least partly), which explains scientific progress [5]. This thesis follows the pragmatic principle in a non-radical manner: although there certainly is a *true* algorithm selection mechanism that maximizes algorithm performance for all future problems, it is merely approximated by generating selection mechanisms and comparing them by their predictive power on some instances. The pragmatic principle motivates an empirical evaluation of different mechanisms, even though their theoretical comparison might be intractable — an approach that is also endorsed by other researchers facing similar problems (e.g., see [172, p. 10]).

1.5. Structure

This thesis consists of three main parts. The first part introduces all relevant concepts and previous work. It outlines relations to other important research questions and surveys existing algorithm selection approaches for applicable solutions. Part two contains the main contribution of this work: it shows how a system for automatic simulation algorithm selection can be built from the theoretical concepts presented in the first part. The resulting methodological framework for simulation algorithm selection has been implemented for the simulation system JAMES II, as a proof of concept. Finally, the third part illustrates how the methods described in part two can be applied to the examples from section 1.3, and investigates the potential benefits of doing so.

Relevant terminology is written in *italics* when being introduced; it can be looked up in the index (p. 260).[2]

[2] Besides terminology, italics are also used to *emphasize* a word, and to demarcate direct quotes. `Monospaced font` refers to software entities, e.g., class names.

Part I.

Background

2. Algorithm Selection

All exact science is dominated by the idea of approximation.
Bertrand Russell

The research domains that are concerned with forms of algorithm selection are diverse and highly differentiated. A reason for this might be the fundamental importance of the problem in many fields of computer science, another one the divergence between theoretical research and practical implementations in the various application areas: *"Many algorithm selection, or parameter tuning, techniques, are tailored to a specific algorithm, and often present similar interesting solutions across different fields of research"* [96, p. 297].

Lacking a coherent terminology, this diversity leads to some confusion and unclarity, which hampers the transfer of algorithm selection solutions among the various domains. As Smith-Miles concludes, *"[...] many related attempts have been made in other disciplines [...], introducing different terminology and overlooking the similarity of approaches"* [292, p. 1], although *"all of these communities would benefit from a greater awareness of the achievements in various cross-disciplinary approaches to algorithm selection"* [292, p. 21].

Therefore, this chapter starts out with the introduction of several fundamental theoretical notions and concepts. The most relevant research areas are outlined, as well as their specific perspectives on the problem. This leads to a catalog of properties to categorize algorithm selection approaches, and hence to reason about their relative advantages and disadvantages in a specific context. Finally, applications of algorithm selection techniques are surveyed and broadly categorized.

2.1. The Algorithm Selection Problem

The *algorithm selection problem* (ASP) has been formally defined by John Rice in 1976 [272]. In its most detailed form, it sheds light on some intricate aspects of the overall problem and also allows to precisely specify some relevant sub-problems. Its formulation, which will be detailed in the following, has been slightly modified and extended to account for concepts from later chapters.

Consider a *problem space* $\mathbb{P}$ containing all possible problems, i.e., input data, on which a set of algorithms operates. The set of algorithms can be regarded as the *algorithm space* $\mathbb{A}$ from which a good algorithm shall be selected. Each problem $x \in \mathbb{P}$ has certain *features*, $f \in \mathbb{F}$, which are retrieved from a problem x via a *feature extraction mapping* $F : \mathbb{P} \to \mathbb{F}$. The extracted features of $x \in \mathbb{P}$ will also be written as f_x ($\in \mathbb{F}$). For simplicity, we can imagine each feature to be encoded as a real number, so that $\mathbb{F} = \mathbb{R}^m$ for m features. It is this *feature space* $\mathbb{F}$ on which the actual *selection mapping* is defined later on, which shall select the best-performing element from the algorithm space $\mathbb{A}$.

Algorithm performance is expressed by a *performance mapping* $p : \mathbb{A} \times \mathbb{P} \to \mathbb{R}^n$, which maps an algorithm $a \in \mathbb{A}$ and a problem $x \in \mathbb{P}$ to an element of the *performance measure space* $\mathbb{R}^n$. Note that the performance measure space has n dimensions, i.e., the performance of an algorithm is multi-faceted. This aspect is quite important, as it accounts for the various metrics of an algorithm's performance: its speed, its memory consumption, its accuracy, and so on. Which algorithm is better — a faster yet less precise one, or a slower yet more accurate one? The definition of 'best' depends on the context in which the algorithm shall be used, in other words the *criteria* of its user. As Knuth nicely puts it when discussing the selection of sorting algorithms, *"[...] there* is *no best possible way to sort; we must define precisely what is meant by 'best', and there is no best possible way to define 'best' "* [181, p. 181]. User criteria can be expressed as a vector $w \in \mathbb{R}^n$ of weights from the *criteria space* and are used to normalize the performance $p(a, x) \in \mathbb{R}^n$ by a norm: $||p(a, x)|| = g(p(a, x), w) \in \mathbb{R}_{\geq 0}$. If, for

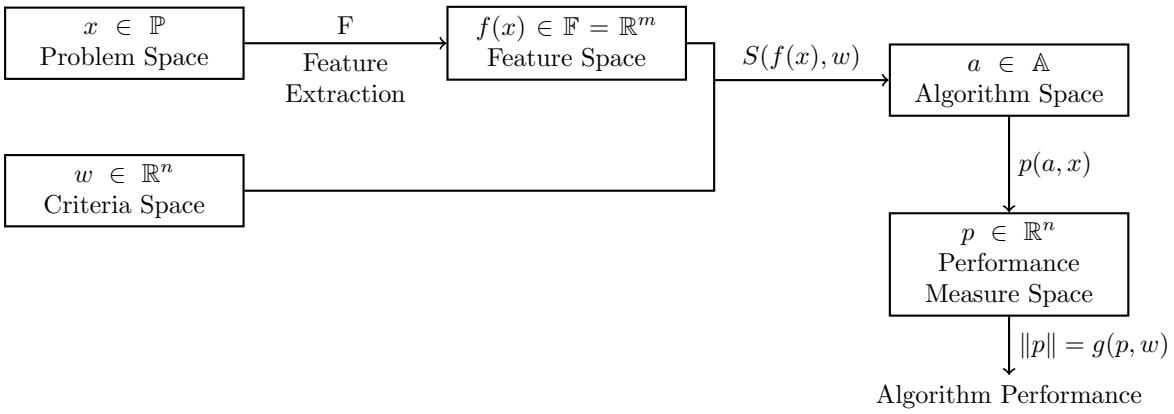

Figure 2.1.: The algorithm selection problem as defined by Rice [272, p. 75].

example, the performance measure space has $n = 2$ dimensions, the first being speed and the second being memory consumption, the user might express with a criterion vector $w = (1, \frac{1}{2})^T \in \mathbb{R}^2$ that speed is twice as important as memory-efficiency in the specific situation.

Consequently, the selection mapping $S : \mathbb{F} \times \mathbb{R}^n \to \mathbb{A}$ considers the features f_x of a problem $x \in \mathbb{P}$ and the user criteria $w \in \mathbb{R}^n$ to select an algorithm $a \in \mathbb{A}$. This completes the overall structure of Rice's formulation of the algorithm selection problem, which is depicted in figure 2.1. Following Rice's notations, one can now formulate the algorithm selection problem as follows:[1]

Definition 2.1.1 (Algorithm Selection Problem (ASP))**.** *Let there be a problem space $\mathbb{P}$, a feature space $\mathbb{F}$, an algorithm space $\mathbb{A}$, a criteria space $\mathbb{R}^n$, a performance measure space $\mathbb{R}^n$, a feature extraction mapping $F : \mathbb{P} \to \mathbb{F}$, a performance mapping $p : \mathbb{A} \times \mathbb{P} \to \mathbb{R}^n$, and a norm $||p(a, x)|| = g(p(a, x), w)$ with algorithm $a \in \mathbb{A}$, problem $x \in \mathbb{P}$, and user criterion $w \in \mathbb{R}^n$. Determine a selection mapping $S : \mathbb{F} \times \mathbb{R}^n \to \mathbb{A}$.*

Note that this problem definition does not specify the *properties* of the selection mapping to be chosen. It merely gives the context of the problem, i.e., all available structures, as well as the basic task, i.e., choosing a selection mapping. What *kind* of selection mapping to be determined is specified by additional criteria. Furthermore, Rice compiled several related problems that arise in this context. The most relevant will be discussed as ASP *sub-problems* in the following.

2.1.1. Important Sub-Problems

The most intuitive and general criterion for choosing a selection mapping S would be to choose one that *always* performs best, so that given all elements from definition 2.1.1 it holds that

$$\forall a \in \mathbb{A}, x \in \mathbb{P}, w \in \mathbb{R}^n : ||p(S(f_x, w), x)|| \geq ||p(a, x)|| \tag{2.1}$$

Is it realistic to search for an all-encompassing selection mapping? Usually, the set of algorithms is already constrained, e.g., by pre-selection from the user. Relevant real-world problems also often lie in a particular *region*, i.e., they are usually not uniformly distributed over the whole problem space $\mathbb{P}$. Due to the potentially high-dimensional spaces of algorithms, criteria, and features, the search for a selection mapping S that complies with equation 2.1 is almost hopeless without reducing the selection mapping search space $\mathbb{S}$, e.g., by predetermining a specific structure for all mappings. For example, searching for the best selection mapping with a given polynomial form restricts the search to the most appropriate coefficients for the prescribed function.

[1] This reflects Rice's basic definition (definition *A*, [272, p. 68]) but extends it by user criteria and feature extraction, which Rice only used in later definitions of ASP sub-problems.

All these points lead to a reformulation of equation 2.1 in that it will be restricted to subsets of problems ($\mathbb{P}_0 \subseteq \mathbb{P}$), algorithms ($\mathbb{A}_0 \subseteq \mathbb{A}$), and selection mappings ($\mathbb{S}_0 \subseteq \mathbb{S}$). Indeed, Rice considered this to be *"perhaps the most realistic situation"* [272, p. 76]. Based on definition 2.1.1, the ASP can now be reformulated as given in definition 2.1.2.

Definition 2.1.2 (Best Selection Mapping Problem (BSMP))**.** *Given the elements from definition 2.1.1 as well as a sub-space of eligible algorithms, $\mathbb{A}_0 \subseteq \mathbb{A}$, a sub-space of relevant problems, $\mathbb{P}_0 \subseteq \mathbb{P}$, and a sub-space $\mathbb{S}_0 \subseteq \mathbb{S}$ of selector mappings to choose from (all of them being defined as mappings $\mathbb{F} \times \mathbb{R}^n \to \mathbb{A}_0$). Choose a selection mapping $S^* \in \mathbb{S}_0$ so that*

$$\forall w \in \mathbb{R}^n, S \in \mathbb{S}_0 : \sum_{x \in \mathbb{P}_0} ||p(S^*(f_x, w), x)|| \geq \sum_{x \in \mathbb{P}_0} ||p(S(f_x, w), x)|| \tag{2.2}$$

This problem will be called the *best selection mapping problem (BSMP)* in the following. Solving it means to find a selection mapping that is best in the sense of definition 2.1.2. The summation of performance over all problems in equation 2.2 relaxes equation 2.1: it is not necessary that the best selection mapping picks the most suitable algorithm for *all* problems, rather it has to deliver best *overall* performance.[2]

Additionally, it might also be necessary to select certain *problem features* that facilitate algorithm selection. These features should allow to *predict* the performance of the considered algorithms as close as possible, i.e., they should convey most information on the algorithm performances. To express this, the set $\mathcal{F}$ of feature extraction mappings $F : \mathbb{P} \to \mathbb{F}$ is introduced. Its elements, the feature extraction mappings $F \in \mathcal{F}$, can be used to segment the problem space $\mathbb{P}$ by an equivalence relation $=_F$:

$$x =_F y \iff F(x) = F(y) \tag{2.3}$$

Let an equivalence class $\epsilon_F(f)$ be defined for the relation $=_F$, a feature extraction mapping $F \in \mathcal{F}$, and features $f \in \mathbb{F}$, i.e.,

$$\epsilon_F(f) = \{x \in \mathbb{P} | F(x) = f\} \tag{2.4}$$

The basic idea is now to search for a feature extraction mapping F that assigns equivalent features to problems on which an algorithm (or a set of algorithms) exhibits rather similar performance. If such a feature extraction mapping could be identified, it helps to *predict* the algorithm performance by identifying all problem features that *influence* it. From an information theory point of view, we look for a mapping F that extracts as much information as possible from all problems, i.e., which reduces the 'noise' of the performance mapping within each equivalence class $\epsilon_F(f)$. For now, the remaining noise is defined as the maximal difference d_F^m for a mapping F, an m-dimensional feature space $\mathbb{F}$, and a set of algorithms A:

$$d_F^m(A) = \max_{f \in \mathbb{F}; a \in A; x,y \in \epsilon_F(f)} ||p(a, x) - p(a, y)|| \tag{2.5}$$

This allows to define the *best features for algorithms problem* (BFAP) as follows:

Definition 2.1.3 (Best Features for Algorithms Problem (BFAP))**.** *Given the structures of definition 2.1.1, a subspace of eligible algorithms $\mathbb{A}_0 \subseteq \mathbb{A}$, and an m-dimensional feature space $\mathbb{F} = \mathbb{R}^m$. Choose a feature extraction mapping $F^* \in \mathcal{F}$ so that:*

$$\forall F \in \mathcal{F} : d_{F^*}^m(\mathbb{A}_0) \leq d_F^m(\mathbb{A}_0)$$

[2] This formulation guarantees that such a selection mapping indeed exists in $\mathbb{S}_0$.

The definition of d_F^m in equation 2.5 is somewhat arbitrary. It minimizes the maximum performance difference of the algorithm given different problems with equal features — but minimizing the *average* difference would also be a reasonable choice. The situation is the same for BSMP (def. 2.1.2): here, the overall performance sum $\sum_{x \in \mathbb{P}_0} ||p(S(f_x, w), x)||$ is used to characterize the performance of the selection mapping S. Another suitable metric would be, for example, to minimize a mapping's worst case performance, $\max_{x \in \mathbb{P}_0} ||p(S(f_x, w), x) - p(S^*(f_x, w), x)||$, with S^* being the best selection mapping.

Generally, such performance metrics can be defined by a *norm* [102, p. 70 et sqq.]: just like the multi-faceted performance $p(a, x) \in \mathbb{R}^n$ is mapped to a single real number by a norm $g(p(a, x), w) = ||p(a, x)||$, a norm can be used to map a vector of performance differences $d_i = ||p(S(f_{x_i}, w), x_i) - p(S^*(f_{x_i}, w), x_i)||$ (for all $x_i \in \mathbb{P}_0$) to a single real number. This number then reflects the performance of the selection mapping S, and can serve as the figure of merit to be minimized. A common form for norms is the so-called L_p*-norm* [102, p. 71], where the d_i are the elements of a vector d:

$$||d||_p = (\sum_i |d_i|^p)^{\frac{1}{p}} \tag{2.6}$$

Three kinds of the L_p-norm are most commonly used in practice:

- L_1*-norm* (or Manhattan norm): The performance differences are simply summed up. For example, the standard deviation σ is related to the L_1-norm, as it is also calculated by summing up the differences.

- L_2*-norm* (or Euclidean norm): This norm represents the Euclidean distance between a selection mapping and an optimal selection, which would be represented by a zero vector: the difference to the optimal performance is zero for all problems. For example, the variance σ^2 is related to the L_2-norm in that the differences between expected and observed values are also squared.

- L_∞*-norm* (or Maximum-norm): The maximum norm maps a vector to its maximal element. It can be regarded as a very conservative approach to maximize a selection mapping's performance, since it is only concerned with the worst case, i.e., the maximal difference to the optimal performance. For any vector x it holds that $L_1(x) \geq L_2(x) \geq L_\infty(x)$ [102, p. 71].

Interestingly, the choice of the specific norm is usually not too important (see discussion in [272, p. 73–75], which also extends the definition of d_F^m to infinite sets). Instead, Rice highlights that *"[...] the crucial ingredients for success are proper choices of the subclasses* $\mathbb{P}_0$*,* $\mathbb{A}_0$*, and* $\mathbb{S}_0$*"* [272, p. 75]. The selection of these three sets are therefore important sub-problems of the ASP. The choice of $\mathbb{P}_0$ is usually motivated by the application context and one merely needs to follow some rough guidelines that take into account the different aspects of the ASP (as described in sec. 7.3.1). Likewise, $\mathbb{A}_0$ is usually predetermined by the set of implemented algorithms in a given system. Hence, the choice of $\mathbb{S}_0$ remains, and it is indeed one of the central challenges: *"[...] the single most important part of the solution of a selection problem is the appropriate choice of the form for the selection mapping"* [272, p. 115]. More light will be shed on this aspect in section 2.1.3. Many of the ideas presented in [272] have been realized in so-called *problem solving environments* (see sec. 2.7). The next sections further develop some of the concepts that stem from this fundamental formulation of the ASP and relate it to other fields.

2.1.2. Effectiveness and Efficiency

As will be seen later, even finding a reasonably good selection mapping for a specific BSMP (see def. 2.1.2) requires some efforts. This section introduces some new concepts that quantify the performance gains of a selection mapping and therefore allow to characterize situations in which algorithm selection pays off. The concepts are based on the basic notions by Rice [272].

At first, it could be asked if there are situations where it is *not* advisable to solve the ASP at all? Clearly, this is the case when an algorithm $a^* \in \mathbb{A}_0$ is known to be *dominant* within the relevant sub-spaces $\mathbb{P}_0 \subseteq \mathbb{P}$ and $\mathbb{A}_0 \subseteq \mathbb{A}$:

$$\forall x \in \mathbb{P}_0, a \in \mathbb{A}_0, w \in \mathbb{R}^n : ||p(a^*, x)|| \geq ||p(a, x)|| \tag{2.7}$$

In other words, an algorithm is dominant if and only if it outperforms (or equals) *any* other algorithm in $\mathbb{A}_0$ with respect to *any* user criteria $w \in \mathbb{R}^n$, and this for *any* problem in $\mathbb{P}_0$. The best selection mapping is easily defined for such situations: $S(f \in \mathbb{F}, w \in \mathbb{R}^n) = a^*$. While discovering the dominance of a single algorithm is usually discouraging for experimental research [163, p. 219], it completely solves the problem of algorithm selection in the given context. Unfortunately, many problem spaces are not dominated by a single (known) algorithm, let alone for all user criteria. Some criteria may even contradict each other, e.g., accuracy and speed.

Now assume that there is no dominating algorithm in $\mathbb{A}_0$. Is solving the best selection mapping problem (def. 2.1.2) always beneficial? Not necessarily. This depends on the context, in particular on the set $\mathbb{S}_0$ of selection mappings that is considered. Let $S_{a'} : \mathbb{F} \times \mathbb{R}^n \to \{a'\}$ be a *constant selection mapping* with $a' \in \mathbb{A}_0$, i.e., a selection mapping that always selects the same algorithm a', regardless of any problem features or user criteria. Imagine a BSMP with $\mathbb{S}_0 = \{S_{a^-}\}$ and a^- being the algorithm with the worst overall performance for given user criteria $w \in \mathbb{F}$:

$$a^- = \underset{a \in \mathbb{A}_0}{\operatorname{argmin}} \sum_{x \in \mathbb{P}_0} ||p(a, x)|| \tag{2.8}$$

Using the best selection mapping from $\mathbb{S}_0$, which is S_{a^-}, would *not* be beneficial at all. In fact, its overall performance for $\mathbb{P}_0$ is even worse[3] than *randomly* selecting algorithms from $\mathbb{A}_0$. To identify selection mappings that indeed outperform the average performance of a random selection, the concept of *average-effectiveness* is introduced. It is based on the notion of selection mapping performance that was already used in equation 2.8 and definition 2.1.2:

Definition 2.1.4 (Overall and Average Performance)**.** *The overall performance $perf(S, P)$ of a selection mapping S on a problem set $P \subseteq \mathbb{P}$ is defined on the structures provided by the BSMP (see def. 2.1.2), including a feature extraction mapping $F \in \mathcal{F}$ and user criteria $w \in \mathbb{R}^n$:*

$$perf(S, P) = \sum_{x \in P} ||p(S(F(x), w), x)||$$

The average performance $\overline{perf}(S, P)$ of a selection mapping S on a problem set $P \subseteq \mathbb{P}$ is defined as:

$$\overline{perf}(S, P) = \frac{perf(S, P)}{|P|}$$

Definition 2.1.5 (Average-Effectiveness)**.** *A selection mapping S^* is average-effective on a problem set $P \subseteq \mathbb{P}$ and an algorithm set $A \subseteq \mathbb{A}$ if (and only if):*

$$\overline{perf}(S^*, P) > \frac{\sum_{a \in A} \overline{perf}(S_a, P)}{|A|} = \frac{\sum_{a \in A} \sum_{x \in P} ||p(a, x)||}{|A| \cdot |P|}$$

Definition 2.1.5 basically ensures that an average-effective selection mapping outperforms the average algorithm performance, i.e., the expected performance when choosing algorithms randomly. Nevertheless, this still does *not* mean that such mappings yield any benefits from exploiting the provided features. Consider the following example, sketched in figure 2.2. Given $A = \{a_1, a_2\}$ and $P = \{x_1, \ldots, x_{200}\}$, with a_1 outperforming a_2 by a rather large margin when applied to $x_1, \ldots, x_{100}$,

[3] Unless all algorithms exhibit the same overall performance.

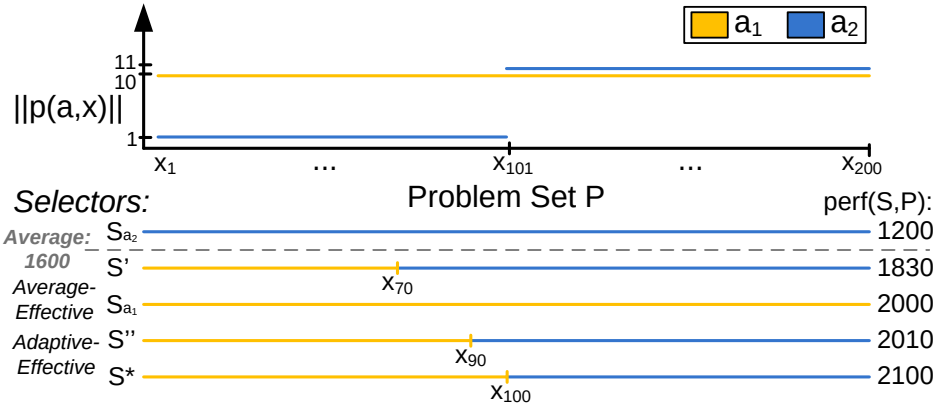

Figure 2.2.: Average-effectiveness versus adaptive-effectiveness of a selection mapping. A list of sample selection mappings is ordered by their overall performance, with S^* being the optimal mapping.

say $||p(a_1, x_i)|| = 10$ versus $||p(a_2, x_i)|| = 1$ for $i \in [1, 100]$. Furthermore, let a_2 perform only *slightly* better than a_1 when applied to $x_{101}, \ldots, x_{200}$, say $||p(a_1, x_i)|| = 10$ and $||p(a_2, x_i)|| = 11$ for $i \in [101, 200]$. Now assume that solving the BSMP led to a selection mapping S' that chooses a_1 for $x_1, \ldots, x_{70}$, and a_2 otherwise. S' is average-effective since (see def. 2.1.5):

$$\overline{perf}(S', P) = \frac{70 \cdot 10 + (30 \cdot 1 + 100 \cdot 11)}{200} = \frac{1830}{200} = 9.15$$
$$\overline{perf}(S_{a_1}, P) = \frac{200 \cdot 10}{200} = \frac{2000}{200} = 10$$
$$\overline{perf}(S_{a_2}, P) = \frac{100 \cdot 1 + 100 \cdot 11}{200} = \frac{1200}{200} = 6$$

and

$$\overline{perf}(S', P) = 9.15 > \frac{\overline{perf}(S_{a_1}, P) + \overline{perf}(S_{a_2}, P)}{|A|} = \frac{10 + 6}{2} = 8$$

However, the constant selection mapping S_{a_1} outperforms S', as the overall performance of a_1 is superior to that of a_2, so that the *average performance* of S_{a_1} for P exceeds that of S': $10 > 9.15$. This means that S' is performing worse than a constant selection mapping that does not *adapt* its decision by considering any problem features. From this it follows that the performance of S' does not justify the prior efforts for feature extraction. To account for this aspect, the notion of *adaptive-effectiveness* is introduced:

Definition 2.1.6 (Adaptive-Effectiveness)**.** *Let $\mathbb{S}_C(A)$ be the set of constant selection mappings for all algorithms in $A \subseteq \mathbb{A}$, i.e., $\mathbb{S}_C(A) = \{S_a | a \in A\}$. A selection mapping S^* is adaptive-effective on a problem set $P \subseteq \mathbb{P}$ if (and only if):*[4]

$$\forall S \in \mathbb{S}_C : \overline{perf}(S^*, P) > \overline{perf}(S, P)$$

If a selection mapping is adaptive-effective, it outperforms any constant selection mapping in $\mathbb{S}_C$, which is only possible by *adapting* the algorithm choice over the problem set P. For example, a

[4]For simplicity, $\mathbb{S}_C(A)$ is often written as $\mathbb{S}_C$; it is usually clear (from the context) to which algorithm set A it refers.

selection mapping S'' that selects a_1 for some more problems than S', namely all $x_i \in P$ with $i \in [1, 90]$, exhibits better overall performance than S_{a_1} and is hence adaptive-effective (cf. fig. 2.2).

All in all, the average-effectiveness of a selection mapping ensures that it outperforms plain guessing, while adaptive-effectiveness means that there is no trivial solution to the problem, i.e., a constant selection mapping, that works better on a given problem set. Consequently, a comparison with a random selection mapping as well as the best constant selection mapping is useful to asses the effectiveness of a BSMP solution. It is easy to show that all adaptive-effective selection mappings are also average-effective, but not vice versa (see previous example and proof on p. 220).

The two levels of effectiveness can be used to test the benefits that stem from algorithm selection in a particular context, but they do not give any idea about the overall *efficiency* of a selection mapping. This requires to put the average performance of a selection mapping into relation with the (hypothetical) maximum performance that can be achieved on a predefined set of problems $P \subseteq \mathbb{P}$:

Definition 2.1.7 (Selection Efficiency)**.** *The efficiency $e(S, P)$ of a selection mapping S on a problem set $P \subseteq \mathbb{P}$ is defined as:*

$$e(S, P) = \frac{perf(S, P)}{perf^*(P)}$$

with $perf^*(P)$ being the maximum overall performance:

$$perf^*(P) = \sum_{x \in P} \max_{a \in \mathbb{A}_0} ||p(a, x)||$$

Since $||\cdot||$ is a norm and by definition $perf^*(P) \geq perf(S, P)$ for any $S \in \mathbb{S}$, it holds that $e(S, P) \in [0, 1]$. The larger $e(S, P)$, the better does the mapping S suit the algorithm selection problem. This allows to express the potential savings of solving a BSMP by calculating the *maximal adaptation gain*, based on the *best constant selection mapping* S_C^*:

Definition 2.1.8 (Maximal Adaptation Gain, Best Constant Selection Mapping)**.** *The maximal adaptation gain for a given BSMP is defined as $1 - e(S_C^*, P)$, with S_C^* being the best constant selection mapping, i.e.,*

$$S_C^* = \operatorname*{argmax}_{S \in \mathbb{S}_C} perf(S, P)$$

This gain quantifies the performance difference between the optimal selection mapping and the best constant selection mapping. The larger the maximal adaptation gain, the larger the potential performance benefits of selecting a suitable simulation algorithm by considering problem features. Similarly, definition 2.1.9 defines the maximal gain to be expected when using the best constant selection mapping. It is the ratio of the best constant selection mapping's performance and the average performance of all algorithms (see definition 2.1.5):

Definition 2.1.9 (Maximal Constant Gain)**.** *The maximal constant gain for a given BSMP is defined as*

$$\frac{|\mathbb{A}_0| \cdot \overline{perf}(S_C^*, P)}{\sum_{a \in \mathbb{A}_0} \overline{perf}(S_a, P)}$$

where S_C^* is again the best constant selection mapping (see def. 2.1.8).

The larger the maximal constant gain, the more performance is gained by identifying the best-suited algorithm and simply selecting it for all problems. Maximal adaptation gain and maximal constant gain therefore help characterizing the benefits of automated algorithm selection under the given circumstances.

2.1.3. Further ASP Properties

Hardness

Guo [116, p. 42–45] illustrates the hardness of the ASP by translating it to a language problem. Given two candidate algorithms $a_1, a_2 \in \mathbb{A}$, described as Turing machines, and a description X of a problem instance $x \in \mathbb{P}$, it is asked if one can construct a Turing machine that decides which of the two algorithms should be selected for solving x. Depending on the user criteria, this could, for example, be the algorithm that returns the correct result with the smallest number of steps. To this purpose, Guo defines a *language of the algorithm selection problem* as

$$A_{TM} = \{(M, X, S, T) | M \textit{ is a TM and accepts } X \textit{ in } T \textit{ steps, returning outcome } S\} \tag{2.9}$$

It can be easily shown that A_{TM} is *recursively enumerable*: the Turing machine to decide the language of the algorithm selection problem just emulates the given Turing machine M on instance X and checks whether it returned the solution S after T steps. Guo argues that the language A_{TM} has a non-trivial property, which corresponds to a non-trivial property of partially computable functions. Here, non-trivial means that at least one partially computable function has this property, but not all of them do. Guo concludes the proof by applying *Rice's theorem*[5], which states that *any* non-trivial property of a partially computable function is undecidable. Presuming that the Church-Turing-Thesis holds, this would mean that there is no algorithm that can tell for two given algorithms and a desired solution which of the algorithms will perform better.

However, the ramifications of Guo's argumentation are limited in case of simulation algorithms, as these usually should halt on any input model. It is quite easy to simulate the operation of two Turing Machines for a *finite* number of n steps by *dovetailing* (see [54, p. 70–85]), i.e., their alternating execution. The upper limit of n required computation steps can be derived by employing *complexity theory* (see sec. 2.2). This allows a comparison of simulation algorithms in terms of speed and solution quality *in theory* — but in practice, this technique is hardly effective: deciding which algorithm is better for a given input by simply trying out all of them will typically result in a large overhead; and since the solution is obtained during the process, there is no need to apply the selected best-performing algorithm to the same input again (unless stochasticity is involved, see sec. 1.3.1).

Relation to Optimization

The principal ASP sub-problem, the BSMP, requires searching for a best element within a given space of selection mappings $\mathbb{S}_0$ (see def. 2.1.2, p. 15). It can therefore be translated into an optimization problem, e.g., in case of a finite[6] set of user criteria $W \subset \mathbb{R}^n$ and a finite set of problems $P \subseteq \mathbb{P}$: Given the entities from definition 2.1.2 and a feature extraction mapping $F \in \mathcal{F}$, find a selection mapping $S \in \mathbb{S}_0$ for which the following objective function $f : \mathbb{S} \to \mathbb{R}_{\geq 0}$ is maximal:

$$f(S) = \sum_{x \in P} \sum_{w \in W} ||p(S(F(x), w), x)|| \tag{2.10}$$

Such a reformulation allows to apply the broad spectrum of techniques that have been developed in the context of optimization theory. Especially algorithms for *black-box optimization*, also called *meta-heuristics*, can be of use in this context, as they do not rely on specific properties of the objective function f, such as differentiability. This is important because the definition of f in equation 2.10 relies on the performance mapping p. Its analytical form is generally unknown (see sec. 2.2, p. 22).

[5] Which is named after Henry Gordon Rice, who conjectured it, not to be confused with John R. Rice, who formulated the ASP. A proof can be found in [54, p. 96].

[6] These assumptions merely serve simplicity, as they avoid the use of integrals within the objective function and reflect a more realistic setup.

Relation to Approximation Theory

Approximation theory is concerned with the approximation of potentially unknown functions by alternative — and usually simpler — functions. An early formulation of the problem was presented by Chebyshev [296, p. viii]: Given a real-valued function $F(x, p_1, \ldots, p_n)$, determine the parameters $\vec{p} = (p_1, \ldots, p_n)^T \in R^n$ so that the maximum error within an interval $[a, b]$ is minimized:

$$\underset{\vec{p}\in\mathbb{R}^n}{\operatorname{argmin}} \max_{x\in[a,b]} |F(x, p_1, \ldots, p_n)| \tag{2.11}$$

Here, the function to be approximated is $f(x) = 0$, so that the maximum error can be regarded as the absolute maximum value of F for a given x and parameters $\vec{p}$. F represents a prescribed *approximation form*, e.g., $F(x, p_1, p_2) = p_1 \cdot x + p_2$ lets all approximations be straight lines in the two-dimensional Cartesian coordinate system. The parameters, or coefficients, p_1 and p_2 are the values that shall be determined. The selection of the coefficients depends on the available information on the function to be approximated, and can in turn be formulated as an optimization problem. The mathematical theory characterizes various classes of approximation forms, e.g., in terms of approximation error. For the ASP, findings from approximation theory can be used to asses the choice of approximation form for the truly best selection mapping, i.e., which $\mathbb{S}_0 \subseteq \mathbb{S}$ is best for a given problem. As explained in section 2.1.1, the proper choice of $\mathbb{S}_0$ is an important sub-problem of the ASP. Rice elaborated on several aspects of approximation theory that might be relevant in the ASP context [272, p. 91 et sqq.], among them:

- Norms and approximation forms
- Convergence and robustness
- Existence and uniqueness

The choice of a norm $||\cdot||$ may have a considerable effect on solving the BSMP, as it is used in the definition of what 'best' is. Nevertheless, Rice concludes from an approximation theoretical point of view that *"[...] the choice of norm is normally a secondary effect compared to the choice of approximation form"* [272, p. 92]. This finding allows to simplify the discussion by defining the norm $g(p(a, x), w)$, with algorithm $a \in \mathbb{A}$, problem $x \in \mathbb{P}$, and user criterion $w \in \mathbb{R}^n$, as a weighted L_1-norm (see eq. 2.6, p. 16) in the following:

$$||p(a, x)|| = g(y, w) = \sum_{i=1}^{n} w_i \cdot y_i \tag{2.12}$$

The question of suitable approximation forms is much more complicated. It will be reconsidered in section 2.3. Convergence, complexity, and robustness are properties that can be evaluated for any kind of approximation form. In the ASP context, convergence refers to a sequence of approximation forms to which the selection mapping sub-spaces $\mathbb{S}_1, \mathbb{S}_2, \ldots \subseteq \mathbb{S}$ correspond. The question is whether this sequence will eventually lead to an approximation form for which the corresponding subset of $\mathbb{S}$ contains the best selection mapping — and if so, how fast this can be expected to happen. For example, such a sequence could be the set of polynomial functions with degree $n = 1, 2, \ldots$ and therefore $\mathbb{S}_1 \subset \mathbb{S}_2 \subset \ldots \subset \mathbb{S}$. The robustness of a selection mapping is a statistical concept that quantifies how much its performance deteriorates when it is applied to unusual selection problems, or when it has been constructed by considering those [121]. Rice illustrates this by comparing arithmetic mean and median, the latter being much more robust than the former when outliers are included [272, p. 100].

Finally, the existence and uniqueness of a best selection mapping can be investigated. The uniqueness is not too relevant from a practical point of view, but the non-existence of effective mappings within a certain sub-space $\mathbb{S}_0$ (in the sense of definitions 2.1.5 and 2.1.6) may lead to deeper insights into the nature of the selection problem at hand.

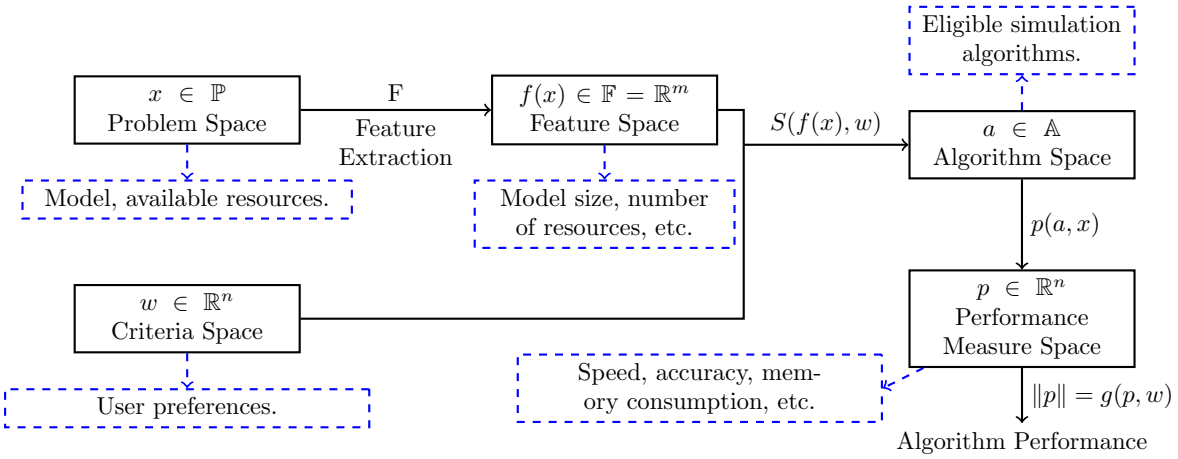

Figure 2.3.: ASP entities and exemplary correspondents from modeling and simulation.

2.1.4. ASP in a Simulation Context

To make use of the theoretical framework described in this section, it is necessary to map it onto the actual problem to be solved here, which is the selection of algorithms for *simulation*. Hence, $\mathbb{A}$ is a set of simulation algorithms that take a model as input and compute the model's behavior. The problem space $\mathbb{P}$ consists of all possible models that can be simulated with the algorithms from $\mathbb{A}$ — but this is not sufficient: as the algorithms can be executed on various computational resources that may strongly influence their performance, resource information is also relevant for algorithm selection and is therefore part of the problem space as well.

Various performance aspects can serve as user criteria. They mainly fall into two categories: On the one hand, there are measurements of *resource consumption*, e.g., of CPU time, network bandwidth, or memory. These shall be minimized. On the other hand, there are measurements of *solution quality*, e.g., accuracy. These shall be maximized. Usually, these two kinds of performance aspects are weighed against each other by the user: either there is a certain degree of solution quality to be reached by employing as little resources as possible, or there is a certain amount of resources that shall be used to calculate a solution that is as good as possible.

Figure 2.3 presents a visual summary. Concrete examples for all the aforementioned entities will be given in section 5.1.1. The following approaches for algorithm selection will be related to the theoretical framework of the ASP, which allows to compare their virtues and shortcomings on a rather abstract yet precise level.

2.2. Analytical Algorithm Selection

A fundamental approach to assess problem hardness and thereby analytically compare the algorithms to solve them is provided by (computational) *complexity theory*. Here, the algorithms $a \in \mathbb{A}$ are compared on a conceptual level, i.e., they are represented by Turing Machines. Depending on the size n of an input problem, complexity theory aims at calculating how much steps and how much memory a specific Turing Machine requires to solve a problem. This is usually expressed asymptotically for different cases by the *Landau Notation*, which uses the following sets of functions [182]:

- $O(f(n)) = \{g(n) | \exists c \in \mathbb{R}, n_0 \in \mathbb{N} : \forall n \geq n_0 : |g(n)| \leq c \cdot f(n)\}$
- $\Omega(f(n)) = \{g(n) | \exists c \in \mathbb{R}, n_0 \in \mathbb{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$
- $\Theta(f(n)) = \{g(n) | \exists c, c' \in \mathbb{R}, n_0 \in \mathbb{N} : \forall n \geq n_0 : c \cdot f(n) \leq g(n) \leq c' \cdot f(n)\}$

These sets reflect the worst, best, and average case performance, respectively. For example, by saying that the memory consumption of an algorithm is in $O(n^2)$, it means that it may grow quadratically with problem size. It does not mean that the memory consumption is *always* quadratic to the problem size — the worst case notation only gives an upper bound. If the memory consumption has the same upper and lower bound, e.g., it is in $O(n^2)$ and in $\Omega(n^2)$, this can be expressed by the average case notation, $\Theta(n^2)$.

This approach has several very important advantages. It allows to abstract away implementation details; these are subsumed by the constants $c, c' \in \mathbb{R}$. All algorithms for one kind of problem are comparable in the most general manner. Using this approach for algorithm selection would mean to choose the algorithm with smallest best-, worst-, or average-case $f(n)$ for a given input size n. Whether to consider memory or time consumption would depend on the user criteria. Additionally, one might derive some estimates of the constants c and c' for each algorithm. This would enable the selection mechanism to use the size n_x of a problem $x \in \mathbb{P}$ as a single feature (i.e., $\mathbb{F} = \mathbb{N} \subset \mathbb{R}^1$) for selecting the most suitable algorithm.

The example illustrates how the greatest strengths of complexity theory — abstractness and generality — also lead to several problems in the ASP context: Firstly, two algorithms may have similar asymptotic behavior but still perform very different, depending on the problem at hand and their specific implementation [146]. This holds for sorting algorithms [181, p. 379–382] and, more important for discrete-event simulation, also for event queues [135]. Another example is matrix multiplication, where the asymptotically best-performing algorithm is not in $O(n^3)$, but in $O(n^{2.8})$: while this is much better in theory, the implementation of the theoretically superior algorithm is non-trivial and merely outperforms the classical algorithm for rather large n [173]. It depends on the implementation and the hardware how large the n has to be.

This points at the second problem of using complexity theory for algorithm selection: it is *very* important to consider the environment the selected algorithm shall be executed in. Some hardware or operating system scales better with certain memory-intensive operations or multi-threading than others, not to mention all specialized hardware that might come into play, e.g., as in [173, 209, 213, 253, 287]. Nowadays, classical complexity theory even fails to predict the fastest algorithm for well-researched problems, such as sorting [191]. This is, for example, because today's hardware architectures incorporate a multi-layered *memory hierarchy*, which is managed by elaborate caching mechanisms. Such hierarchies typically consist of registers, several cache levels, random access memory, and swap space on the hard disk. The time required by each of these components for read/write access varies by several orders of magnitude. Therefore, algorithms that avoid *cache misses*, i.e., the situation when a cache does not contain the required data any more and a lower (thus, slower) level of the memory hierarchy has to be queried, are preferable over those that deliver best performance *in theory* [191]. Caching is itself a hard problem for which various heuristics might be applied, as discussed in [255].

A third problem with using complexity theory for algorithm selection is its inability to consider more problem features, and not just the feature extraction mapping $F(x) = |x|$ that yields the input size. As Fellows points out, *"[...] real inputs are not random, but rather have lots of hidden structure, that may not have a familiar name, even if you knew what it was"* [78, p. 298]. It is this hidden structure that gets (at least partly) unveiled by more suitable feature extraction mappings from $\mathcal{F}$ in the ASP (see def. 2.1.3), but is generally not accounted for in complexity theory. In [117], this problem is described as moving from problem complexity to problem instance complexity.

Finally, complexity theory is a challenging discipline, i.e., it may require considerable effort to prove any bounds on time or memory consumption. For example, problems occur when analyzing the performance of self-organizing sequential search algorithms, so that McGeoch concludes: *" [...] any single analytical model will give an incomplete picture of an algorithm's performance"* [227, p. 199]. The analytical efforts for algorithm comparison cannot be automated, and developers of new simulation algorithms do not necessarily have a strong background in complexity theory.

However, it has to be said that theoretical computer scientists are quite aware of the shortcomings that hamper the wide application of classical complexity theory in practice [325], and therefore complement their work more and more with empirical studies [163, 235]. Such studies on *empirical*

hardness (e.g., [205]) are promising because of the so-called *phase transitions* [37] that some problem domains exhibit (see sec. 2.7, p. 56).

Furthermore, the model of the Turing Machine can be extended to include some of the aspects that the classical theory abstracts away. Abstract machine models have been proposed that resemble modern computers in that they have RAM, are networked with other machines to exchange messages, and are interacting with a user [315]. Other approaches even extend the original definitions toward quantum computers [58] or biologically inspired computing [266]. Besides worst and average case analysis, Guo mentions the analysis of algorithm performance on certain sub-classes $\mathbb{P}_0 \subset \mathbb{P}$ [116, p. 20]. Rice hints at some problems regarding the characterization of selection mappings that could be investigated by complexity theory [272, p. 95]. Complexity theory can also be enhanced by a more sophisticated consideration of problem features. This is done in the field of *parameterized complexity theory*, where a parameter $k \in \mathbb{N}$ is given in addition to the problem size n, i.e., $\mathbb{F} = \mathbb{N}^2$. This allows, for example, to specify more precise worst case bounds for some well-known theoretical problems [78].

Still, it seems to be clear that complexity theory is not suitable for *automatically* solving any algorithm selection problem *in practice*. Its fundamental contribution to the understanding of problem hardness and the limits of algorithm performance are nevertheless very valuable, and allow developers to select the most promising algorithms for implementation. In this sense, complexity theory forms the implicit basis of all automated algorithm selection methods, as it is often used for manual algorithm pre-selection.

2.3. Algorithm Selection as Learning

Complexity theory bases its analysis on the abstract definition of the algorithms in $\mathbb{A}$. Another possibility is to restrict the analysis on a finite set of *empirical* performance data

$$(f_{x_1}, a_1, p_1), \ldots, (f_{x_i}, a_j, p_k), \ldots, (f_{x_l}, a_m, p_n)$$

with $f_{x_i} \in \mathbb{R}^m$ being the extracted features of a problem $x_i \in \mathbb{P}$, and $p_k \in \mathbb{R}^n$ being the observed performance of algorithm $a_j \in \mathbb{A}$ when applied to problem x_i. These tuples will be called *performance tuples*:

Definition 2.3.1 (Performance Tuple, Performance Tuple Set)**.** *A tuple $\phi = (f,a,p) \in \mathbb{F} \times \mathbb{A} \times \mathbb{R}^n$ is called a performance tuple (for a problem $x \in \mathbb{P}$, with $F(x) = f$). Its elements may be denoted as f^ϕ, a^ϕ, and p^ϕ, respectively. A set $\Phi = \{\phi_1, \ldots, \phi_n\}$ is called performance tuple set.*

The empirical data contained in a performance tuple set Φ allows to investigate several important aspects of the ASP, e.g., the magnitude of noise caused by a poor feature extraction mapping or a poor experimental setup. It can be regarded as an empirical estimation of d_F^m (see eq. 2.5, p. 15), this time on the grounds of a performance tuple set Φ:

$$\max_{\phi_1,\phi_2 \in \Phi \wedge (f^{\phi_1}, a^{\phi_1}) = (f^{\phi_2}, a^{\phi_2})} ||p^{\phi_1} - p^{\phi_2}||$$

Moreover, approximation theory can be applied to the given data in Φ: Is it possible to construct a good approximation function $F(x, p_1, \ldots, p_n) \mapsto \mathbb{A}$ (see eq. 2.11, p. 21)? How large will be the error, i.e., the deviation from the best possible algorithm selection? How to find the best parameters p_i, and which approximation forms are suitable?

Approximation theory strives for general answers to these questions and is often focused on certain classes of functions — but it can also be regarded as the foundation of two more practical disciplines [49]: *statistical learning* [122] and *machine learning* [333]. Statistical learning, also referred to as *learning theory*, forms the theoretical base of machine learning. Machine learning also covers the computational aspects of learning, i.e., algorithms that allow an efficient generation and evaluation of the approximation forms motivated by statistical learning theory. Other practical aspects, such as the preprocessing of data, are also part of machine learning methodology [333, p. 247 et sqq.] but

typically not considered by learning theory. In conjunction, both fields offer a wide range of sound practical methods to generate, parameterize, and evaluate approximation functions for data observed in the real world. For the ASP, such data could have the form of performance tuple sets.

Learning theory allows to find a suitable approximation function for the performance of a, i.e., a function that *predicts* the algorithm's performance for new problems that exhibit similar features. Having such *performance approximation functions* $\widehat{perf}_a : \mathbb{F} \to \mathbb{R}^n$ for all algorithms in $\mathbb{A}_0$ allows to define a selection mapping

$$S(f, w) = \underset{a \in \mathbb{A}_0}{\operatorname{argmax}} ||\widehat{perf}_a(f)|| \tag{2.13}$$

The search for such a selection mapping can be regarded as an approximation problem, similar to that described in equation 2.11 (p. 21). We search for some parameters $\vec{p} \in \mathbb{R}^k$, with which S can be adjusted so that it exhibits a minimal maximum error when compared to the maximal performance (see def. 2.1.7, p. 19):

$$\underset{\vec{p} \in \mathbb{R}^k}{\operatorname{argmin}} \max_{x \in \mathbb{P}} ||perf^*(\{x\}) - p(S(F(x), w, \vec{p}), x)|| \tag{2.14}$$

Equation 2.14 basically gives an alternative formulation of the best selection mapping problem from definition 2.1.2 (p. 15)[7], whereas equation 2.13 suggests a possible form for S, i.e., to construct it from individual performance approximation functions $\widehat{perf}_a$ for all $a \in \mathbb{A}_0$. If this form is chosen, the approximation problem from equation 2.14 is not concerned with the specific parameters of S anymore, but with the parameters $\vec{p} = (\vec{p_1}^T, \ldots, \vec{p_k}^T)^T$, with $k = |\mathbb{A}_0|$ and $\vec{p_i} \in \mathbb{R}^{k_i}$. The k_i-dimensional parameters $\vec{p_i}$ are used to parameterize the corresponding performance approximation function for algorithm a_i, i.e., $\widehat{perf}_{a_i} : \mathbb{F} \times \mathbb{R}^{k_i} \to \mathbb{R}^n$:

$$\forall a_i \in \mathbb{A}_0 : \underset{\vec{p_i} \in \mathbb{R}^{k_i}}{\operatorname{argmin}} \max_{x \in \mathbb{P}} ||p(a_i, x) - \widehat{perf}_{a_i}(F(x), \vec{p_i})|| \tag{2.15}$$

The approximation problems defined in equations 2.14 and 2.15 are fundamentally different from a learning theory perspective: the approximation function S in equation 2.14 has to choose an element from $\mathbb{A}_0$, a set that can be assumed to be finite in practice, whereas the performance approximation functions $\widehat{perf}_{a_i}$ from 2.15 have a range of $\mathbb{R}^n$. In learning theory, the approximation of a function with a finite range is called *classification*, while *regression* denotes the approximation of functions with a continuous range. In both cases, the most important step is to actually *relate* the problem features f_{x_i} to algorithm performance — either by predicting the performance of each algorithm and choosing the one with best predicted performance (regression, see eq. 2.15, p. 25), or by classifying which algorithm will be the best for the problem at hand (eq. 2.14).

From a human perspective, the generation of good approximation functions by a program would appear to a user as if the program would be *learning*, since it finds the approximation functions by considering the empirically generated performance tuple set. The performance tuple set can be regarded as past experience in executing the algorithms on certain problems. This is why the theory of (statistical) learning is so intimately connected to approximation theory [49], and therefore provides a multitude of techniques that can be exploited to solve the ASP.

2.3.1. Error Sources, Error Types, and the Bias-Variance Trade-Off

A central aspect of learning theory is to estimate and analyze the error of using a learned, i.e., approximated, function S instead of the best selection mapping S^*. As already discussed for definition 2.1.3 and d_F^m (eq. 2.5, p. 15), various reasonable measures can be used to assess the performance of a selection mapping. In learning theory, such a typical *loss function* l is the squared error loss, which can be defined for $F(x)$ and S as follows:

[7] Instead of maximizing the overall performance (def. 2.1.2, p. 15), equation 2.14 minimizes the maximal error — see discussion of norms in section 2.1.1 (p. 16).

$$l(y,(x,w)) = ||(y - p(S(F(x),w),x))||^2$$

where $y = perf^*(\{x\})$ is the optimal performance (see def. 2.1.7, p. 19) and the tuple $(x,w) \in \mathbb{P}\times\mathbb{R}^m$ contains the input for the selection mapping S, i.e., the problem $x \in \mathbb{P}$ for which the algorithm shall be selected and some user criterion $w \in \mathbb{R}^m$. Based on such a loss function, statistical learning theory allows to analyze the *expected prediction error (EPE)*, i.e., the overall error to be expected when applying S to all problems in $\mathbb{P}$, as a function of S (see discussion in [122, p. 18–22]). If the expected prediction error for any given selection mapping S can be calculated, this would make the search for a good selection mapping considerably easier, since its future performance could be estimated. Unfortunately, such an error analysis is non-trivial and hampered by several aspects.

Firstly, the set $\mathbb{S}$ of *all* possible selection mappings is limited to a subset $\mathbb{S}_0 \subset \mathbb{S}$ in practice, similar to the definition of the best selection mapping problem (def. 2.1.2, p. 15). This is usually done implicitly by defining the structure of S, for which the best parameters are then searched (e.g., eq. 2.14, p. 25). In learning, $\mathbb{S}_0$ is often referred to as the *hypothesis space* and the approximation form (see sec. 2.1.3, p. 21) is called a *model*. Following this nomenclature, the approximation form $f(x) = \alpha \cdot x + \beta$ would be a model of all linear functions, with α and β being its parameters. To avoid confusion with simulation models, such models will explicitly called *approximation models* in the following. Each $S \in \mathbb{S}_0$ can be regarded as a hypothesis with respect to the suitability of the available algorithms for given features $f \in \mathbb{F}$ and user criteria $w \in \mathbb{R}^m$. The restriction of the hypothesis space to $\mathbb{S}_0$ introduces the so-called *approximation error*, or *bias*, since the overall best selection mapping S^* is not necessarily in $\mathbb{S}_0$. In other words, the bias stems from the selection of an approximation form.

Secondly, the performance tuple set Φ is finite and does not contain data on all elements of $\mathbb{P} \times \mathbb{A}$. This introduces the *sample error*, or *variance*, which causes the choice of sub-optimal parameters for the selection mapping S, since they are chosen to approximate the sample data only. This error type reflects the fundamental problem of inductive knowledge, as discussed in section 1.4.

Finally, the performance of an algorithm for a problem with given features f is usually not deterministic. It has to be regarded as a random variable that underlies an unknown probability distribution. Hence, there is an *irreducible error* that cannot be avoided, even if S^* is used. The stochasticity of algorithm performance measurements is caused by three error sources:

- Variation on lower abstraction levels: The algorithms to be tested are working on top of several abstraction levels of hard- and software. They are executed within a specific runtime environment, which in turn runs on a specific operating system. Eventually, all instructions of the algorithms have to be processed by the hardware, which again exhibits several abstraction levels with complex interactions, e.g., between the CPU and its caches [191]. All these influences could be explicitly resolved in theory, but in practice it is much easier to regard the effects that let execution time vary for the same sequence of instructions as random [123]. Due to the *central limit theorem* [277, p. 70–71], this noise can be approximated by a normal distribution. All performance measurements regarding the consumption of computational resources, e.g., CPU time, memory, or network bandwidth, are susceptible to this kind of error.

- Variation in $\mathbb{P}$: As motivated for the BFAP sub-problem (def. 2.1.3, p. 15), it is important to select suitable problem features for extraction. The maximal difference of two algorithm performance measurements for problems with identical features f should be minimal, which is reflected in the definition of $d_F^m(\mathbb{A}_0)$ in equation 2.5. If $d_F^m(\mathbb{A}_0) > 0$, there are still some problem aspects that are not expressed by any feature, although they influence algorithm performance. The variation of algorithm performance due to these problem aspects can be treated as additional noise with an unknown probability distribution [114]. All kinds of performance measurements are influenced by this source of stochasticity, which highlights the importance of selecting the most suitable features for problem discrimination.

- Problem-inherent variation: if simulation models contain stochastic elements, their execution may lead to different trajectories. The performance measurements of the algorithm simulating

any of these trajectories may be strongly influenced by the properties of the specific trajectory. For example, consider the model of a biochemical system that evolves with a probability of 0.9 to an oscillating system, or else it remains in equilibrium. An algorithm that performs well on oscillating models would succeed in 90% of the simulation replications, but its performance could be much worse for the other 10% of possible trajectories. This issue pertains all performance measurements of stochastic simulations.

Not even the best selection mapping can overcome these three sources of algorithm performance variation. To overcome the first cause of variation, it would require to consider the *whole* state of all hard- and software that is involved, which is unrealistic. The second and the third causes cannot be distinguished by the procedure for learning a selection mapping: since problems are reduced to their features, it is obscured whether performance varies per problem or per class of problems with identical features.

Note that the stochastic noise introduced by the three aforementioned sources of variation is only irreducible from a learning perspective. To reduce it in practice, it might be necessary to minimize the influence of uncontrolled variables on performance measurements while experimenting (e.g., by locking out other users or shutting down background jobs like virus scanners), to identify new problem features, or to reduce the randomness of the model behavior (see sec. 3.2.1, p. 66).

The overall expected prediction error EPE consists of the three kinds of error discussed above — bias, variance, and the irreducible error. Formally, the expected prediction error $EPE(x)$ for a selection mapping S, a problem $x \in \mathbb{P}$ and user criteria $w \in \mathbb{R}^m$ can be written as [122, p. 197]:

$$EPE(x) = Bias^2(S(x,w)) + Var(S(x,w)) + \sigma_\epsilon \tag{2.16}$$

where the mathematical expressions for $Bias$ and Var depend on the form of S [122, p. 196 et sqq.]. Since σ_ϵ is irreducible, the only way of reducing the EPE is to reduce the error due to bias and variance. Unfortunately, it turns out that there is a trade-off between both error types, called the *bias-variance trade-off*. Reducing the error from predefining the model structure, i.e., the bias, usually makes a learning method more susceptible to errors in the sample, i.e., it increases its variance: if the hypothesis space is enlarged, it is likely that a hypothesis can be found that better fits the sample data. Hence, a bias reduction can be regarded as allowing a prediction function to fit the sample data better and better. This might lead to *overfitting*, when the function is geared so much to the sample data that prediction suffers because it wrongly relies on irrelevant aspects. Here, the sample variance dominates the error due to bias. On the other hand, restricting the hypothesis space too much will lead to functions that cannot be fitted well to the sample data, and therefore fail to make good predictions because of a too simplistic structure that is not able to capture the essential associations between input and predicted output. This phenomenon is known as *underfitting*, and applies in cases where the bias dominates the variance.

Two simple examples for the different bias and variance characteristics of learning schemes are depicted in figure 2.4. In both cases (upper and lower row), a value shall be predicted by considering a one-dimensional input. In the first row (case A and B), the input and the values exhibit a relationship that is almost linear. With linear regression, one now chooses the coefficients α and β for the function $f(x) = \alpha \cdot x + \beta$ so that the error on the sample data (green dots) is minimized, thereby constructing an approximation function as shown in case A. Since linear regression has a low variance, using other data as sample data would have resulted in very similar choices of α and β, i.e., very similar functions. The *one-nearest neighbor* (1-NN) approach basically memorizes all points from the sample data, and then predicts the value of a new input by looking up the value from the sample data with the most similar attributes, i.e., its single closest neighbor in the input space[8]. In the first example, the predictions of the one-nearest-neighbor approach (case B, dotted lines) are much worse than that of linear regression (A). Note that the prediction quality of 1-NN could be strongly improved by adding more points to the sample data — in contrast to linear regression, where these additions would

[8] In practice, nearest-neighbor approaches usually take more than just one neighbor into account. This is just a special case of the so-called k-nearest-neighbor methods (k-NN).

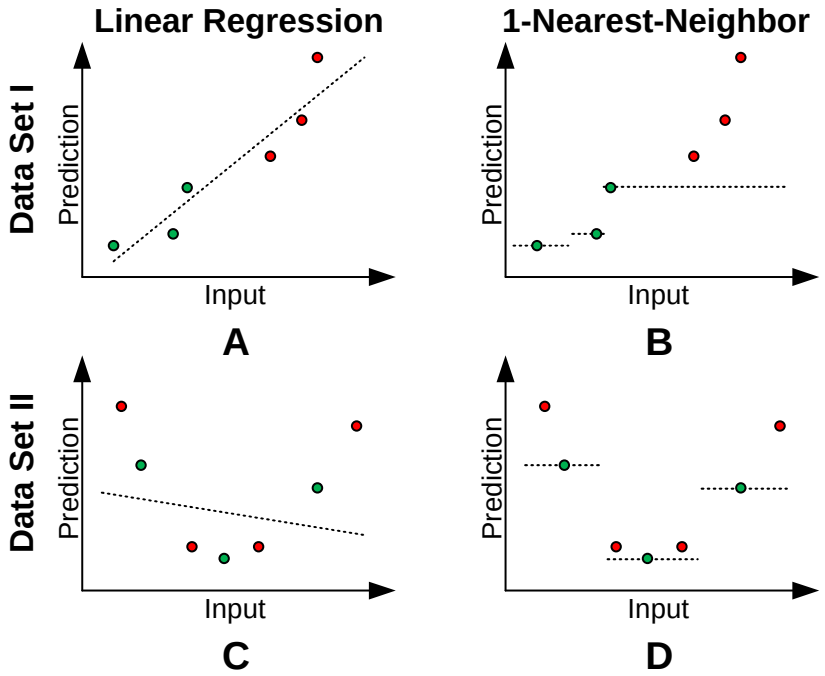

Figure 2.4.: Prediction of linear regression (case A and C) and the one-nearest-neighbor approach (case B and D), compared on two data sets (cases A/B and C/D). Green dots denote sampled elements from which the prediction functions would have been constructed, whereas red dots show additional points of the underlying data distributions. The predictions of the two methods are sketched by dotted lines.

not have such a strong impact on the coefficients. This illustrates the higher variance of 1-NN when compared to linear regression, i.e., its prediction quality is more dependent on the sample data. In the second example, the strong bias of linear regression has a negative impact on its prediction quality; a linear relationship between input and prediction is now presumed wrongly (case C). Here, the predictions of 1-NN are better (case D), since it has a low bias and can thus be fitted to data of various shapes. Note, however, that the high variance of 1-NN *still* may lead to bad predictions, e.g., when only relying on the three sample points in the very middle of the parabola.

To minimize the *overall* expected prediction error (EPE), it is necessary to find a good balance between bias and variance. This is highly problem-specific and also depends on the method for generating the selection mapping S. Moreover, it requires additional techniques for *estimating* the expected prediction error — just considering the *resubstitution error*, i.e., the error of the prediction function when confronted with the sample data that were used to construct it in the first place, is much too optimistic (e.g., see [122, p. 200 et sqq.] or [333, p. 121]). By steadily increasing the complexity of the approximation form, i.e., enlarging the hypothesis space, one will eventually find a hypothesis that just memorizes the value to be predicted for each sample data. Its resubstitution error therefore equals the irreducible error σ_ϵ. Clearly, such a prediction function would perform very badly on any new data — so the key idea for EPE estimation is to divide the sample data into test and training data sets, to fit the prediction function to the training data, and to then observe the error of the prediction function on the formerly *unseen* test data[9]. Such an EPE estimate would not be overly optimistic, as it is now evaluated how the prediction function reacts to 'new' data. Figure 2.5 illustrates why a training sample should not be used for estimating the EPE, and how the bias-variance trade-off makes finding a good approximation form essentially a search problem.

[9]The unseen data can be further split into a test set and a validation set. The latter is used to finally validate the performance of the prediction function that performed best on the test set [122, p. 196].

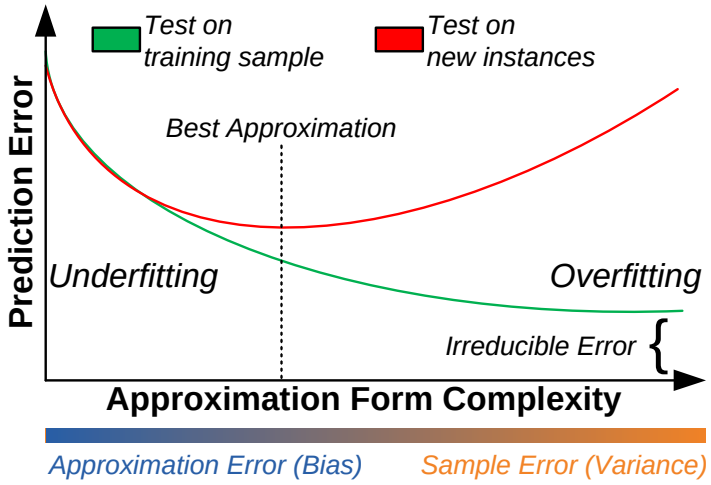

Figure 2.5.: An illustration of the bias-variance trade-off, inspired by [122, p. 194].

2.3.2. Reinforcement Learning

The learning problem discussed so far relies on abundant sample data, e.g., in form of performance tuples (see def. 2.3.1, p. 24), which is used to find an approximation function. This kind of learning could be regarded as *"learning with a teacher"* [172, p. 6], as the prediction function is only confronted with problems for which the outcome is already known. This way of learning is therefore known as *supervised learning*. However, there are many natural learning processes that do not work this way. Instead of predicting function values and comparing the prediction with the actual outcomes provided by a supervisor, animals and humans often learn by trial-and-error: they choose to perform some action and receive some kind of feedback, or *reward*, that reflects the appropriateness of their choice. Positive rewards *reinforce* an agent, i.e., an entity that is *acting* in an environment, in its choice of action, whereas negative rewards bear the incentive to choose differently next time. The goal of the agent is to maximize its *overall* reward.

This form of *unsupervised learning* is called *reinforcement learning* [279, p. 763 et sqq.] and can be regarded as *"learning with a critic"* [172, p. 6]. Instead of using some algorithm to find the best-fitting approximation form parameters for some sample data, an agent uses reinforcement learning to predict which action to take, given the perceived state of its environment. The accuracy of this prediction cannot be observed directly, only by considering the returned reward. Hence, reinforcement learning basically classifies the states of the given environment with respect to the action that is currently most beneficial in terms of overall reward, without necessarily knowing anything about the given environment from the onset.

Reinforcement learning is inherently *incremental* in that it discovers step by step which rewards the actions yield in which situation. Consequently, it is particularly suitable for environments that are prone to stochastic effects. Even if this is not the case, many other difficult situations, e.g., faulty sensors to perceive the environment or faulty effectors to conduct actions, can be effectively modeled by *assuming* that the environment behaves randomly to some extent (see discussion in [172, p. 17–22]). Stochasticity requires the agent to find a good balance between the *exploration* of its environment, e.g., by trying out new actions to take, and the *exploitation* of the knowledge that was already acquired by former trials. If too much sub-optimal actions are wasted on re-assuring that they are indeed sub-optimal in a given situation, i.e., the agent was not just 'unlucky' in receiving a low reward for choosing them, the *overall* reward of the agent will deteriorate [96, 171]. Similarly, overly exploitative strategies will not yield optimal reward, as they may be ignorant of better actions that have not been explored thoroughly. In principle, this well-known *exploration vs. exploitation trade-off* is about balancing the optimistic view that a better-performing action can be found against the

pessimistic view that no such better action exists. A compromise between both extremes is strongly problem-dependent and should also change with time, as the agent's knowledge of its environment increases.

Markov Decision Processes

The underlying theory of reinforcement learning is based on *Markov decision processes (MDPs)*, which in this context are usually restricted to be *finite* [298].[10] A finite *Markov process* is a stochastic process with a *state space* $Z = \{z_1, \ldots, z_n\}$ for which the Markov property holds, i.e., it is memoryless in that the transition probabilities to future states only depend on the current state $z \in Z$. A Markov decision process extends this concept by a finite set of *actions* $C = \{c_1, \ldots, c_m\}$. Before each state transition, it now has to be decided which action from C shall be carried out. The selected action shall influence the process in some desirable manner, i.e., steer it towards desirable states. The desirability of a state $z_i \in Z$ is expressed by associating it with a certain *reward*, which can be defined as a function of the states [279, p. 615] or as a *reward distribution*, so that the reward of deciding for action $c \in C$ in state $z \in Z$ and thereby reaching state $z' \in Z$ can be expressed as an expected value [298, p. 66]:

$$R^c_{z,z'} = E\{r_{t+1} | z_t = z \wedge z_{t+1} = z' \wedge c_t = c\}$$

where r_{t+1} denotes the reward received after the $(t+1)$-th state transition and c_t is the action chosen before. Likewise, z_t and z_{t+1} are the states after the t-th and $(t+1)$-th state transitions. To completely determine the reward structure of the MDP, the actual reward distributions $\mathcal{R}^c_{z,z'}$ (with $R^c_{z,z'}$ as their expected value) have to be defined for all $c \in C$ and $z, z' \in Z$. Additionally, the state transition probabilities $\mathcal{P}^c_{z,z'}$ have to be defined for all combinations of $c \in C$ and $z, z' \in Z$ [298, p. 66]:

$$\mathcal{P}^c_{z,z'} = Pr\{z_{t+1} = z' | z_t = z \wedge c_t = c\}$$

The transition probabilities may depend on the current state z and also on the action c that was chosen, but not on past states or actions. The notion of time is expressed by the index t, so that the process evolves over time in a discrete-stepwise manner. If all $\mathcal{R}^c_{z,z'}$ and $\mathcal{P}^c_{z,z'}$ are constant over time, i.e., independent of t, the MDP is said to be *stationary*. In contrast, *non-stationary* MDPs express circumstances that are changing while decisions are made. Research on Markov decision processes dates back to the 1950s [18], since they are a suitable model for many domains that involve some form of decision making, particularly for resource allocation and consumption problems [330].

The role of a decision-maker for an MDP is carried out by a so-called *policy*. It can be regarded as the strategy with which an agent chooses actions and interprets rewards, i.e., its method of reinforcement learning. Strategies can be formulated mathematically or as algorithms. They have the same objective like the agents in reinforcement learning, namely to maximize their overall reward.[11] The definition of the overall reward depends on the *horizon*, i.e., the number of action-reward rounds that shall be considered. If a strategy's overall reward for an MDP is measured over a finite horizon, the received rewards can just be summed up. In case the MDP has an infinite horizon, summing up sub-optimal rewards will still result in infinite sums, which are hard to compare. Here, the overall reward is usually *discounted*, e.g., *geometrically*: each received reward is multiplied by γ^n, where $\gamma \in (0, 1)$ is the *discount factor* [109, p. 14] and n is the number of the round that is currently played; γ^n approaches zero when n approaches ∞. This allows to limit the reward sums to finite amounts and hence makes them comparable. Although sometimes only used because of tractability considerations [16], introducing a discount factor also makes sense in some real world settings, e.g., to express interest rates. When sufficient information on a given MDP is available, an optimal MDP policy can be computed, e.g., by *dynamic programming* [44, p. 323 et sqq.]. Generally, the complexity

[10] A more formal definition of continuous MDPs can be found in [109, p. 13 et sqq.].

[11] Although in the context of optimization, the equivalent problem of minimizing the *cost*, instead of maximizing the reward, is discussed as well [248].

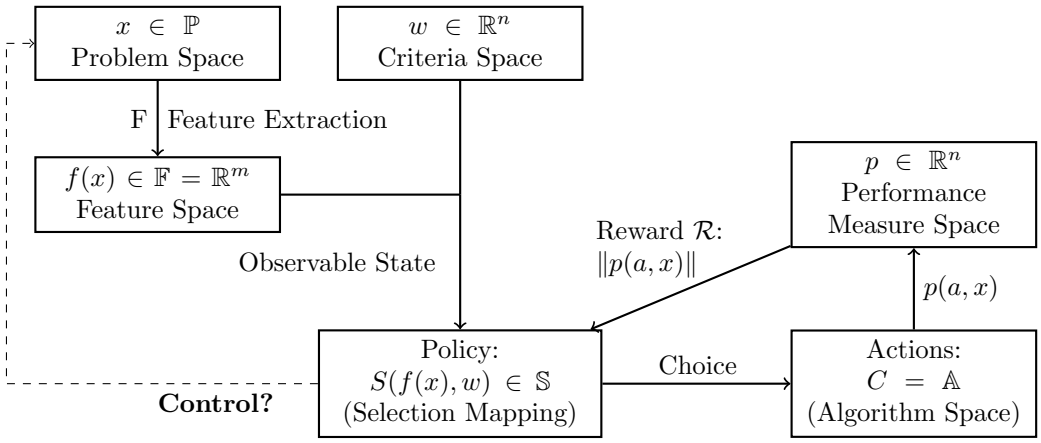

Figure 2.6.: The entities of the algorithm selection problem in the context of a Markov decision process (cf. fig. 2.3). The dotted arrow denotes the lacking influence of the policy's choice on the MDP state, i.e., the simulation problem and the user criteria.

of algorithms which 'solve' an MDP problem, i.e., decide whether there is a policy that achieves a certain amount of reward, depends on the stochasticity and the observability of the state space [248]. Anyhow, in many real-world settings the information required by these algorithms, i.e., $\mathcal{P}^c_{z,z'}$ and $\mathcal{R}^c_{z,z'}$, is not available. These are the settings in which reinforcement learning can be applied to incrementally approximate an optimal policy by empirically estimating the nature of the environment (i.e., $\mathcal{P}^c_{z,z'}$) and also the most rewarding course of action (i.e., $\mathcal{R}^c_{z,z'}$).

Algorithm Selection as a Markov Decision Process

To model the algorithm selection problem with a Markov decision process, the set C of actions to choose from could be regarded as the set of algorithms $\mathbb{A}$. Since a selection mechanism will be continuously invoked to decide on a good algorithm for a particular simulation problem and user criteria $w \in \mathbb{R}^n$, the problem space $\mathbb{P}$ and the criteria space $\mathbb{R}^n$ would be a good starting point to construct the state space Z of the process. However, a selection mapping $S \in \mathbb{S}$, which would be in the role of a policy as it selects the action (algorithm) for the current state (simulation problem), is defined on the *features* $f_x = F(x)$ of a problem $x \in \mathbb{P}$. Hence, only the features f_x are observable for the policy S — although the MDP's state space Z comprises $\mathbb{P}$. The state space can be reformulated as $Z = \mathbb{R}^n \times \mathbb{P} \times \mathbb{F}$, but now the policy may only observe the state's features, i.e., the state is only *partially observable*.[12] Partially observable states make an MDP problem hard to solve [248].

Figure 2.6 summarizes the construction of a Markov decision process from ASP entities graphically. The reward distributions can be related to the performance of the selected algorithm $a = S(f_x, w)$, with user criteria $w, w' \in \mathbb{R}^n$, simulation problems $x, x' \in \mathbb{P}$, and features $f_x, f_{x'} \in \mathbb{F}$:

$$\mathcal{R}^a_{(w,x,f_x),(w',x',f_{x'})} = ||p(a,x)|| = g(p(a,x),w) \tag{2.17}$$

As discussed in section 2.3.1, several important performance metrics, such as execution time or memory consumption, are influenced by factors that are hard or impossible to control. They should therefore be represented by random variables that are drawn from specific probability distributions, i.e., the $\mathcal{R}^a_{(w,x,f_x),(w',x',f_{x'})}$.

However, equation 2.17 suggests that the performance of a on problem x does *not* depend on the next state $(w', x', f_{x'})$ that is reached. Clearly, this is determined either by the user submitting a new problem, or by an algorithm that generates new simulation problems on the fly (e.g., for optimization,

[12] The inclusion of features does not add any information, but makes the state space dependent of a feature extraction mapping $F \in \mathcal{F}$, so that $(w, x_i, f_{x_i}) \in Z \iff x_i \in \mathbb{P} \wedge f_{x_i} = F(x_i)$.

see sec. 3.2.2, p. 68). Here, Markov decision processes — and hence, reinforcement learning — address a harder problem setting, in which an agent is able to *influence* its environment with its actions, whereas it merely has to react to it in a sensible manner for solving the ASP (see fig. 2.6).

The reward distributions of an MDP for algorithm selection are therefore reduced to $\mathcal{R}^a_{w,x,f_x}$, and the estimation of any state transition probability $\mathcal{P}^a_{(w,x,f_x),(w',x',f_{x'})}$ is futile, as selecting an algorithm should not have any impact on the next simulation problem or user criteria. This shows that the general reinforcement problem is harder to solve than an algorithm selection problem in the sense of section 2.1.[13] It is therefore beneficial to consider simplifications of reinforcement learning, such as the multi-armed bandit problem.

The Multi-Armed Bandit Problem

A widely known reinforcement learning problem refers to the special case where the underlying MDP has just one state: $Z = \{z_0\}$. Here, the agent is not able to perceive any distinct states of its environment, so that the MDP's transition model is trivial, i.e., $\mathcal{P}^c_{z_0,z_0} = 1$ for all actions in C. The agent merely needs to learn about the unknown reward distributions $\mathcal{R}^c_{z_0}$ associated with its choice of action $c \in C$, so that the notation of reward distributions can be abbreviated to $\mathcal{R}_c$, with expected values R_c. All problem features are neglected, and it is further assumed that the user criteria $w \in \mathbb{R}^n$ are constant and already included in the reward calculation.

Interestingly, this problem is closely related to the (statistical) field of *sequential experiment design* [274] (see sec. 3.2), where it is known as the *multi-armed bandit problem* (MABP) and is investigated since the early 1930s [305]. Even at that time, when the problem had not yet been generalized to reinforcement learning or connected to MDPs, it was still regarded as related to *"[...] the general question of how we learn — or should learn — from past experience."* [274, p. 530].

The term multi-armed bandit problem stems from an illustrative real-world example, in which a gambler plays against a generalization of the 'one-armed bandit' known from casinos, which now has k arms instead of just one. Each arm has an associated probability distribution for reward, but this distribution is unknown to the player. The player shall now maximize the reward over a number of iterations, repeatedly choosing one of the k arms and examining the returned reward. Policies to play this game therefore aim at approximating the underlying reward distributions $\mathcal{R}^c_{z_0}$ empirically. Due to its simplicity, the MABP *"[...] provides the natural setting for studying the trade-off between exploitation and exploration [...]"* [170, p. 517] — which is a central issue in reinforcement learning.

Even in this simplified setting, finding an optimal policy is non-trivial. In [109], Gittins shows how to construct optimal policies for discounted MABPs. The basic idea is to associate each arm with an allocation index, which can be deduced from prior information. However, there are some practical limitations to Gittins' approach: the indices may be hard to calculate and optimality only holds for geometrically discounted problems [16, p. 5–6]. Furthermore, the method assumes that *"[...] the player can compute ahead of time exactly what payoffs will be received from each arm, and their problem is thus one of optimization, rather than exploration and exploitation"* [10, p. 323]. Such information is not available in many practical scenarios, which call for heuristic policies instead [11, 316]. While heuristic policies are not guaranteed to always choose the optimal action, some can be shown to *converge* to an optimal behavior when infinitely many rounds are played, i.e., in case of an infinite horizon. This is done by analyzing the *regret* of a policy, which denotes the expected reward difference between using a given policy and an optimal choice. Auer et al. formally define the regret of a policy on a k-armed bandit after n rounds as [11, p. 236]:

$$R^* \cdot n - \sum_{i=1}^{k} R_{c_i} \cdot E[T_i(n)]$$

[13] While the above derivation of an MDP for algorithm selection is straightforward, other formulations have been proposed, e.g., [188, 256], that do take into account state transitions. They focus on a repeated algorithm selection during the execution of a single problem (see sec. 2.5.3).

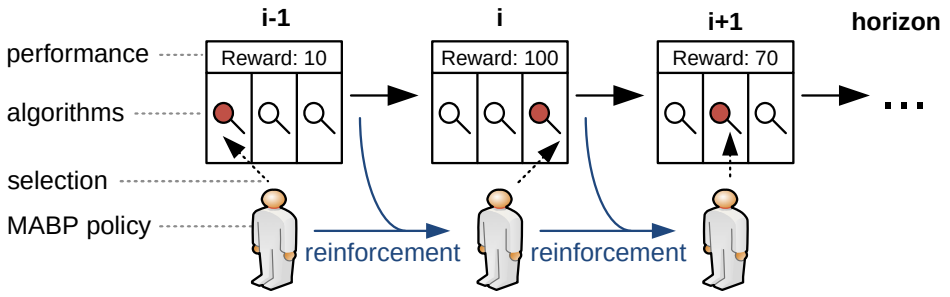

Figure 2.7.: The multi-armed bandit problem as an algorithm selection problem

where R_{c_i}, $i = 1, \ldots, k$ is the expected reward of distribution $\mathcal{R}_{c_i}$ associated with arm i, $R^* = \max_i R_{c_i}$ is the expected reward of the best arm, and $E[T_i(n)]$ is the expected number of times the policy chooses arm i during the n rounds. Note that the reward distributions, i.e., the $\mathcal{R}_{c_i}$, are implicitly assumed to be stationary (i.e., they do not depend on n), since the underlying MDP would have to have more than one state otherwise. However, many policies can be adjusted to non-stationarity reward distributions by weighting recent and past rewards differently [298, p. 38].

Policies whose regret approaches zero for $n \to \infty$ are called *zero-regret* policies. Intuitively, a zero-regret policy is able to recover from any *lock-in* due to unfavorable random reward outcomes, i.e., a situation in which the unlikely high rewards of choosing a sub-optimal arm lead to a neglect of exploring better alternatives (see [170, p. 517]). Theoretical analysis of the MABP provides a general lower bound on a policy's regret [190], and some upper regret bounds could be found for specific policies (e.g., [11]).

In an algorithm selection context, each arm of the bandit could again represent an eligible algorithm, i.e., $k = m = |\mathbb{A}| = |C|$, as already discussed for MDPs in general (see sec. 2.3.2, p. 31). Likewise, the underlying probability distributions would correspond to the stochastic reward in terms of delivered performance (see fig. 2.7). Generally, policies for MDPs can be regarded as *search algorithms* for an optimal selection mapping S^*. As a special case, policies for the multi-armed bandit problem conduct this search without considering any features: $\mathbb{F} = \mathbb{R}^1$ and $F(x) = 0$ (or some other arbitrary number). They do not distinguish between different simulation problems, i.e., states of the MDP. While it therefore seems overly optimistic to apply MABP policies to the ASP in general, they are still suitable to handle specific situations, e.g., when $\mathbb{A}$ contains a dominating algorithm (see eq. 2.7, p. 17).[14]

The notion of regret is closely related to the selection efficiency of the found mappings S (see def. 2.1.7, p. 19); zero-regret policies will eventually identify S^* and use it often enough to let the policy's regret approach zero for $n \to \infty$. From this perspective, it is not only important that S^* is eventually identified, but also *how fast* a policy is able to do so. The convergence speed of policies is therefore an important aspect, which is typically investigated by empirical studies [11, 316].

The multi-armed bandit problem can be extended in several ways. Instead of just being confronted with the *partial information* of the chosen arm's reward, some settings may provide *full information*, i.e., the rewards of all k arms for each round [10]. Other scenarios abandon the assumption of constant distributions and even consider an adversary player, which considers a policy's behavior and distributes the reward in a way that the policy's overall reward gets minimized [10]. The plain and simple MABP as discussed above is important in many application domains, e.g., for constructing clinical trials [170].

2.3.3. Further Aspects of Learning

Machine learning subsumes a large interdisciplinary body of methods, with many dimensions that may serve for categorization. From an agent perspective [279, p. 37], a central distinction is that of

[14] Here, $\forall f \in \mathbb{F}, w \in \mathbb{R}^n : S^*(f, w) = \{a\}$, with $a \in \mathbb{A}$ being the dominating algorithm.

online versus *offline learning*, i.e., whether an approximation form is found before runtime (offline) or during runtime (online). Particularly the latter scenario motivates the use of *incremental learning*, i.e., methods that allow incremental changes to the approximation form (e.g., version spaces [279, p. 683]). They do not require to store the complete training set of past data, as this is already encoded in the approximation form to be adjusted. Many reinforcement learning techniques work this way, as the current reward will merely be used to update the estimated value of the performed action and can therefore be dismissed afterwards. The distinctions between classification and regression (see sec. 2.3) or supervised and unsupervised learning (see sec. 2.3.2) are somewhat orthogonal to the above notions, although some combinations prevail: for example, the unsupervised techniques from reinforcement learning usually work online. Such fundamental properties help to categorize ASP solution approaches, as discussed in section 2.6.

The field of learning seems to provide the most promising techniques to solve the ASP. One form of algorithm selection mapping already suggested by Rice in [272] strongly resembles decision trees, the base for a nowadays popular family of approximation forms [269, 333]. Anyhow, the *learnability* of a selection mapping that is effective (see sec. 2.1.2, p. 16) is not guaranteed and depends on the computational complexity of the learning algorithm [313] as well as the amount of training data. The computational complexity of learning relates to the so-called *metareasoning-partition problem* (e.g., [116, p. 39]): it is usually hard to balance between the effort for learning *how* to solve a problem and the effort to actually solve it. For the ASP, metareasoning subsumes the effort to generate a suitable selection mapping, including the collection of training data, error estimation, and so forth. This effort needs to be amortized by the savings that the continued application of the generated mapping brings about. The problem is related to the measures of effectiveness defined in section 2.1.2 (p. 18): only adaptive-effective selection mappings warrant the effort of metareasoning via problem feature extraction at all.

The bias-variance trade-off discussed in section 2.3.1 shows that there is no silver bullet to learning. It therefore seems unlikely to identify a learning method that works best for all kinds of algorithm selection problems — instead the extensive literature on the relative merits of different approaches should be considered for case-by-case guidance (e.g., [20, 122, 172, 279]). The literature also links learning techniques to other related fields, e.g., meta-heuristics for black-box optimization. As outlined in section 2.1.3, the ASP can be solved by optimizing a partly unknown objective function $f(S)$. The black box in this case is the performance mapping p (see eq. 2.10, p. 20), i.e., our ignorance of the exact relation between algorithm, input problem, and algorithm performance. From a learning perspective, an meta-heuristic optimizer therefore *learns* which selection mapping S is likely to perform best if the given training data is representative for future problems. Wolpert and Macready proved *no-free-lunch theorems* for optimization algorithms [334], which essentially state that all optimization algorithms perform equally well when the *whole* range of optimization problems is considered. This motivates a careful selection and adaptation of different methods to solve specific ASPs.

2.4. Algorithm Selection as Adaptation to Complexity

From an outside perspective, one could say that a simulation system which automatically selects algorithms to solve simulation problems *adapts* itself to its environment. While this view is similar to reinforcement learning (see sec. 2.3.2), it is not restricted to it — focusing on *adaptation* as such reveals additional challenges, aspects, and approaches to a solution.

2.4.1. Complex Simulation Problems

Adaptation is strongly related to the notion of *complexity*. Regardless of algorithmic complexity theory, which merely characterizes the execution time and memory requirements of a given algorithm (as discussed in sec. 2.2), a general definition of complexity is quite hard to come by and often depends on the entities to be considered. One fundamental approach to formally define and quantify complexity has been put forward by Kolmogorov and others in the context of *information theory* [46,

p. 463 et sqq.]. The *Kolmogorov complexity* $K_U(x)$ of a string x denotes its *descriptive* complexity, by defining it as the minimal size of a program that prints x and then halts when executed on a universal computer[15] U [46, p. 466]:

$$K_U(x) = \min_{p|U(p)=x} l(p)$$

where p is a program for U and $l(p)$ is its length. Kolmogorov complexity is *noncomputable* [46, p. 483], i.e., no Turing Machine can calculate it for an arbitrary x in finite time, and is therefore only of limited use in practice. Still, it reveals many interesting connections between the notion of complexity in philosophy, physics, and computer science. For example, it underpins the validity of *Occam's razor* [46, p. 488], a fundamental principle of induction (see sec. 1.4, p. 8) that demands to choose the simplest possible explanation for a phenomenon. Similarly, Kolmogorov complexity is strongly related to the notion of *entropy*, which connects thermodynamics and information theory [80, p. 137 et sqq.], in that a truly random sequence x_r has maximal entropy, which means it cannot be compressed by any algorithm, and hence $K_U(x_r) \geq l(x_r)$: put simply, no program p describing x_r can be more elegant than saying `'print` x_r`'` [46].

While associating complexity with completely random sequences is sometimes desirable, Edmonds [61] points out that this notion is somewhat counter-intuitive: adding interrelations between the elements of a sequence reduces the amount of information it contains, i.e., its entropy, but should not make it *less* complex. Instead, he proposes to define complexity as *"that 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"* [61, p. 6]. This definition shifts the focus from descriptive to behavioral complexity, and is therefore quite relevant for discussing the complexity of models to be simulated. In the sense of this definition, a model (which can be regarded as the description of a system in a formal modeling language, see sec. 1.2) is complex if its *overall behavior* cannot be formulated easily, so that simulation is required in the first place. The unknown model behavior, in conjunction with the complex hardware it shall be simulated on,[16] also makes it unclear which simulation algorithm to use. Hence, the environment to which a simulation system has to adapt consists of *complex simulation problems*.

Note that a solving a single simulation problem may require multiple replications, in case stochasticity plays a role (see sec. 2.3.1, p. 25). To avoid any confusion regarding terminology, the problem solved by a single simulation run is referred to as a *simulation problem instance* whenever necessary. In the following, it is generally assumed (without loss of generality) that all instances of a simulation problem $x \in \mathbb{P}$ have the same problem features $f_x \in \mathbb{F}$.

2.4.2. Complex Adaptive Systems

The relation of adaptive behavior and complexity is twofold: one the one hand, successful operation in complex environments often requires adaptive behavior, e.g., by means of reinforcement learning (see sec. 2.3.2). On the other hand, adaptivity is often realized by complex mechanisms. This interconnectedness of complexity and adaptation becomes apparent when considering *complex adaptive systems (CAS)*, i.e., systems that are *both* complex and adaptive. Such systems often exhibit similar properties, e.g., in terms of their internal network structure [15]. They can be found in various domains, e.g., in biology (human brain, ant colonies) or socio-economics (cities, companies).

Holland regards adaptation as the driving force behind ever more complex biological organisms [141], and models CAS as multi-agent systems (see [141, p. 41 et sqq.] and sec. 1.3.2, p. 6). Each agent's behavior is governed by a set of rules that are strengthened whenever the agent receives a *reward* at the time a rule was active and hence influenced the agent's behavior. A rule is activated when its context matches the current situation of the agent and it succeeded in a bidding process that depends

[15] A computer that has the same computational abilities as a Turing Machine (see sec. 1.2).

[16] Today's hardware is complex in the same sense as the model: it is hard to formulate overall behavior (e.g., how long does a cache miss take?) although every detail of the system is known.

on its current strength. This procedure can be regarded as reinforcement learning on a set of rules, and hence yet another example of implicit inductive reasoning (see sec. 1.4): if a rule contributes to gaining rewards, it gets strengthened. Holland sees the rules as hypotheses with respect to the agent's best course of action. New rules are discovered by using a *genetic algorithm*: the best-performing rules are selected and recombined to create a new generation of rules. The new rules get mutated randomly, which helps to discover unprecedented variants.

With these two techniques — multi-agent systems in combination with genetic algorithms — Holland attempts to built a simple model that shows phenomena similar to those found in CAS, including the emergence of new behavior caused by adaptation. As the term *genetic algorithm* suggests, he draws his inspiration from evolutionary processes in biology (e.g., [55]), as biological organisms are the most prominent example of CAS in nature. In [140, p. 28], Holland formally defines an adaptive system as a tuple $(\mathfrak{A}, \Omega, I, \tau)$, with $\mathfrak{A}$ being the set of all possible system structures, Ω being the set of (probabilistic) operators defined upon $\mathfrak{A}$, and I being the set of possible inputs to the system. The *adaptive plan* $\tau : I \times \mathfrak{A} \to \Omega$ selects an operator for adaptation, based on current input and system structure. As the selected operator $\omega \in \Omega$ is then used to determine a new system structure from $\mathfrak{A}$, the adaptive plan basically steers the structural evolution of the system over discrete time steps. This adaptation is governed by system input and the randomness of the operators in Ω. To solve the best selection mapping problem (def. 2.1.2, p. 15), for example, one could define an adaptive simulation system as the tuple $(\mathbb{S}_0, \Omega, \mathbb{P}_0, \tau)$. It would react on the input in form of simulation problems (from $\mathbb{P}_0$) by adapting its current selection mapping, i.e., choosing another element from $\mathbb{S}_0$. An adaptive plan τ can now be used to learn the best selection mapping incrementally. Holland bases his formulation of genetic algorithms[17] on these entities. However, it is uncertain if such a general and incremental mechanism is the right choice to tackle the ASP: both $\mathbb{S}_0$ and $\mathbb{P}_0$ are usually quite large, and users may fail to issue the amount of simulation problems that is necessary for convergence.

Biological CAS have also been studied from a mathematical (e.g., [276]) and a physical perspective, where their existence can be related to the same laws of thermodynamics that are also linked to information theory's notion of complexity [284]. Apart from understanding how CAS originated, investigating them might yield important insights when it comes to designing artificial CAS with similar properties, such as adaptiveness. For example, genetic algorithms are nowadays a widely applied meta-heuristic, also used in many settings that are not related to biology (e.g., in parallel simulation [323]). Therefore, it is worthwhile to discuss which CAS principles have already been applied in software systems, and might hence enable simulation software to adapt itself to its current environment, characterized by the (usually complex) simulation problems it is confronted with.

2.4.3. Self-Adaptive Software and Autonomous Computing

In the domain of software architectures, self-adaptivity has mainly been discussed in the context of *middleware* [230], which refers to the software layer that manages the interaction between applications and the operating system they run on. A middleware typically offers functionality that is shared by a broad class of applications, e.g., network communication, resource management, or service discovery. An adaptive middleware could therefore reconfigure itself to best serve the needs of the current applications, or to react to changes in its environment, e.g., CPU load or network connectivity. Although the implementation of suitable decision making mechanisms is regarded as a key challenge in itself [230], middleware research focuses on designing software architectures in which it is easy to make such informed decisions and to apply them.

McKinley et al. distinguish between *compositional adaptation* and *parameter adaptation*, the former term referring to changes in the *structure* of the software, whereas the latter means to merely alter parameter values. They classify adaptive software approaches by three aspects: *how* composition is done, *when* it is done, and *where* it is done [230]. The first aspect subsumes various software patterns and programming language features that allow composition, e.g., wrappers or function pointers. The

[17] Interestingly, he shows the effectiveness of genetic algorithms by relating them to the multi-armed bandit problem discussed in section 2.3.2 (p. 32) [140, p. 125 et sqq.].

second aspect relates to the time at which composition is possible, e.g., compile time or run time, while the last aspect specifies the software layer affected by the composition, e.g., operating system or middleware. Furthermore, they highlight the importance of approaches that provide compositional adaptation, and name *separation of concerns*, *reflection*, and *component-based design* as essential techniques for implementing them.

Separating concerns and making software components re-usable are commonplace design principles for many kinds of (non-adaptive) software. Component frameworks have been used for over a decade to construct problem solving environments [99], which is a central application domain for algorithm selection methods (see sec. 2.7). Reflection, i.e., the access of a software system to its own structure, ensures that the system becomes self-aware to some extent [22]. This *self-awareness* provides the information on which the decisions for re-composition are made.[18] Reflection may either be *structural* or *behavioral* [22, 230]: the former allows to inspect the internal (sub-)structures of a given component, while the latter allows to inspect its runtime behavior, i.e., its performance. Since algorithm selection often relies on empirical performance data, this makes behavioral reflection an essential part of a software system supporting algorithm selection, while structural reflection might not always be required.

Very similar to self-adaptive software systems are so-called *autonomous computing systems (ACS)*, a term coined by a corresponding IBM research program [153]. Such systems are defined to have the following features:

- **Reflection**, i.e., ACS are self-aware.
- **Perception**, i.e., ACS are aware of their context.
- **Continuous Optimization**, i.e., ACS incrementally improve their operation.
- **Robustness**, i.e., ACS are self-healing in case of malfunction.
- **Self-Adaptivity**, i.e., ACS are able to reconfigure themselves.
- **Usability**, i.e., ACS hide complexity from the user.
- Security, i.e., ACS perform self-protection.
- Open Interfaces, i.e., ACS comply with open, heterogeneous environments.

The **boldly** printed characteristics are all in some way related to the ASP. Reflection and perception are required to reason on the algorithms in $\mathbb{A}$ and to extract the features $f_x \in \mathbb{F}$ from the current simulation problem $x \in \mathbb{P}$. Continuous optimization is a desirable feature; it can be implemented by any *incremental* learning scheme, e.g., reinforcement learning (see sec. 2.3.2). Increasing the robustness, self-adaptivity, and usability of a simulation system is what any solution to the ASP should deliver: robustness is enhanced if the re-selection of algorithms is triggered by failures, the system adapts itself to simulation problems, and the overall automation should increase usability. Hiding the *complexity* from the user is regarded as *"[...] the ultimate goal of autonomic computing [...]"* [153, p. 28] — the ultimate goal of algorithm selection is to hide the complexity of deciding upon the most suitable simulation algorithm.

In this sense, algorithm selection supports the development of autonomous simulation systems that comply to all of the above requirements. While the security of simulations is usually handled by the infrastructure, e.g., operating systems and network equipment, open interfaces are also a desirable feature and a prerequisite for joint research. Nevertheless, algorithm selection as such does neither require nor improve the openness of a system.

[18] The components for decision-making could in turn be subject to reflection and compositional adaptation *themselves*, etc., which results in an infinite regress — the term usually indicates that one part of the system is aware of *another* part.

Software engineering is struggling with the realization of ACS features for many years. For instance, *self-adaptive software* can be implemented by communicating software agents [187, 177]. Laws et al. follow this direction by proposing agent models for such systems [194] and highlighting links to cybernetics, complex adaptive systems, and biologically inspired algorithms [141, 193]. Cybernetic research provides some fundamental insights into adaptive systems. For example, Laws et al. argue with Ashby's *Law of requisite variety* [193] that any system that controls another system has to be at least as complex — in terms of its number of states — as the system on which it exerts control. From this, one could deduct that any optimal algorithm selection approach has to be at least as complex as the minimal description of performance differences between all the algorithms from which it shall select. Although this strong interpretation of Ashby's law is rejected in [128] by an intuitive counterexample, a cybernetic perspective suggests that hiding *more* complexity from the user requires *more* complex ACS. Likewise, the laws of *requisite* and *incomplete* knowledge [128] highlight the importance of knowledge on the system in order to control it, and the fact that this knowledge is necessarily incomplete. Apart from incomplete knowledge, all adaptation mechanisms also face the metareasoning-partition problem (see sec. 2.3.3, p. 33). In simulation, this trade-off becomes apparent with adaptive methods for speeding up parallel and distributed simulation, such as load balancing (see sec. 1.3.2). Ignoring this trade-off may lead to *worse* performance, especially if the adaptation scheme is unsuitable for the problem at hand and is computationally expensive itself [71].

To tackle these issues, it could be helpful to consider complex adaptive systems from nature, e.g., Blair et al. mention them as future research directions for their adaptive middleware OpenORB [22], which supports component-based design and reflection. Karsai et al. argue for a control engineering perspective and propose an additional supervisor component on top of the normal system: *"On the ground-level one can create components that are highly optimized for specific situations, while the supervisory-level will have to recognize what situation the system is in, and select the most optimal component"* [176, p. 28]. A single software agent is also used in [60] to realize *self-adaptive numerical software (SANS)*, which adapts to problems from scientific computing. Armstrong et al. propose a component architecture specifically designed for high-performance computing, motivated by the various algorithms and mechanisms involved in the large-scale distributed simulation and visualization of a combustion engine [8]. Some simulation systems claim component-based design and adaptivity as key features, but often this merely means that earlier design decisions do not *prohibit* such mechanisms. One example for this is *DEVS/RMI*, a tool for distributed simulation of DEVS models [340], which "*[...] is also built to support auto-adaptive [sic] and reconfiguration of simulations during runtime*" [341, p. 1]. This is done by implementing the migration of model and simulation entities over a network, and by providing a central simulation controller along with monitoring services. While these features are necessary for some kinds of adaptation and reconfiguration, they still only *enable* those mechanisms — they do not realize them.

2.5. Algorithm Portfolios

Another area concerned with an informed selection from a set of available options is *portfolio theory* in economics, where *securities* (e.g., stocks) are aggregated into a *portfolio* with certain desirable properties. A portfolio can be regarded as a weighted set of securities, where the weights sum up to 1 and denote the relative amount of capital invested in each one. In other words, a portfolio is a specific linear combination of securities. The assessment of portfolios has a long history in financial analysis. In [215], Markowitz investigates the hypothesis that an investor should always strive for the portfolio with the maximal expected return. He argues that this maxim does not consider the value of *diversification*, i.e., the reduction of risk by distributing the capital over several securities with risks that do not correlate. The values (and hence the risk) of securities can correlate heavily, e.g., stocks of two companies in the same market may both loose their value if the market shrinks.

Consequently, *portfolio selection* should consider both, the expected return E and its statistical

variance V, which Markowitz calls the *E-V rule*.[19] He identifies a set of *efficient* portfolios, i.e., portfolios for which no alternative portfolio with at least the same expected return and *less* variance exists, or with at most the same variance and *more* expected return. Such a set of efficient portfolios is depicted in figure 2.8. Since it is always at the border of the overall portfolio set when plotting it to an *E-V* diagram, it is also called the *efficient frontier* [114]. Portfolios that are elements of the efficient set are also called *efficient portfolios*. It is left to the investor to decide upon the most suitable portfolio from the efficient frontier, i.e., the best trade-off between expected return and the investment risk, the latter being represented by the variance.

Finally, it should be noted that Markowitz' theory aims at *explaining* why investors were *already* building diversified portfolios that did not yield a maximum expected return, i.e., portfolio theory is essentially concerned with *"[...] the analysis of the real world phenomenon of diversification"* [43, p. 26]. Diversification is also known to be an important aspect of adaptation [141], and hence of complex adaptive systems as discussed in section 2.4. In this sense, portfolio theory provides decision guidelines for dealing with the complexity of economic systems. In fact, the notion of portfolio optimality can also be discussed from an information theoretical point of view (see sec. 2.4), where continuous re-investment of the capital in considered. This leads to the notion of *log-optimal portfolios* [46, p. 613 et sqq.], i.e., portfolios that are optimal when their successive returns are multiplied.

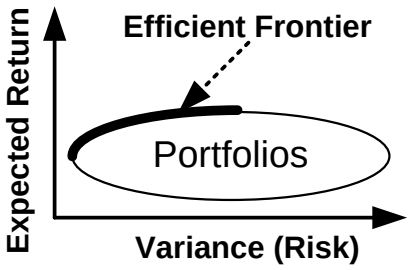

Figure 2.8.: The efficient frontier.

2.5.1. Identifying Efficient Portfolios

The basic idea of constructing financial portfolios, as put forward in [215], is to estimate expected return and variance for a single portfolio from past stock data. Markowitz also suggests to let experts provide corrections and adjustments for the data. The returns of stocks $x_1, \ldots, x_n$ are regarded as a set of random variables $R_1, \ldots, R_n$. Returns are paid out at a regular frequency, e.g., yearly. Let the past returns previously earned by stock x_i be $r_1^{x_i}, \ldots, r_k^{x_i}$, i.e., these are samples from the random variable R_i.

A portfolio is defined as $\vec{\alpha} \in [0,1]^n$ with $\sum_{i=1}^{n} \alpha_i = 1$, where α_i denotes the share of the capital that is assigned to asset x_i, and therefore has to be in $[0,1]$. The expected value of portfolio $\vec{\alpha}$'s return $R_{\vec{\alpha}}$ can be calculated easily:

$$E[R_{\vec{\alpha}}] = \sum_{i=1}^{n} \alpha_i \cdot E[R_i] \tag{2.18}$$

Since stock prices are influenced by general economic developments and their corresponding companies also compete with each other, the random variables $R_1, \ldots R_n$ cannot be assumed independent. Therefore, calculating the variance of $R_{\vec{\alpha}}$ needs to take into account the *covariance* $\sigma_{i,j}$ between returns R_i and R_j:

$$\sigma_{i,j} = \rho_{i,j} \cdot \sigma_i \cdot \sigma_j \tag{2.19}$$

where σ_i and σ_j are the standard deviations of R_i and R_j, respectively, and $\rho_{i,j}$ is the *correlation coefficient* for R_i and R_j.

[19] His approach is also called mean-variance portfolio selection, as it focuses on these aspects of the overall portfolio performance.

Then, the variance of portfolio $\vec{\alpha}$ can be written as:

$$V[R_{\vec{\alpha}}] = \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot \sigma_{i,j} \tag{2.20}$$

Using equations 2.18 and 2.20 allows to estimate the variance and the expected return of any portfolio $\vec{\alpha}$ by estimating the expected values ($\hat{R}_i$) and standard deviations ($\hat{\sigma}_i$) for all R_i, as well as the correlation coefficients for all pairs of R_i and R_j.[20] These estimations are based on the sample data $r_1^{x_1}, \ldots, r_k^{x_n}$.

While the above formulas estimate the merits of a specific portfolio, the problem of *finding* the set of all *efficient* portfolios is solved by Markowitz only for the case of three securities, via analytic geometry [215]. More generally, the problem can be defined as a *quadratic programming problem*, i.e., an optimization problem with an objective function in which the parameters to be optimized (the α_i) are multiplied with each other. Markowitz' basic approach can therefore be formulated as maximizing $f(\vec{\alpha})$ [234]:

$$f(\vec{\alpha}) = \lambda \sum_{i=1}^{n} \alpha_i \cdot \hat{R}_i - (1-\lambda) \cdot \sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \cdot \alpha_j \cdot \hat{\sigma}_{i,j} \tag{2.21}$$

where $\hat{R}_i$ are the estimations of the expected returns, $\hat{\sigma}_{i,j}$ are the estimations of the covariances (see eq. 2.19, p. 39), and $\lambda \in [0, 1]$ determines the desired trade-off between risk and return (as decided by the investor).

Adding cardinality constraints, i.e., limiting the size of the portfolio (the number of $\alpha_i > 0$) to a certain interval, makes this a *quadratic mixed-integer problem*, which is hard to solve [234]. Metaheuristics for optimization like genetic algorithms can be successfully applied in such cases, to find almost optimal portfolios [331].

There are some objections to following such portfolio selection approaches. It has been shown by resampling experiments that the efficient portfolio selected by this method might still turn out to be sub-optimal, as it is fitted against (potentially insufficient) past data [165]. This is yet another manifestation of the problem of inductive knowledge, as discussed in section 1.4 (p. 8). Furthermore, it could be argued that the economy is governed by power laws, which render the focus on mean and standard deviations troublesome, as low-probability events have a major impact on the overall outcome but are not sufficiently included in risk analyses [300, p. 277 et sqq.]. Alternative formulations that do not consider the return's variance as a measure of risk, but instead focus on the absolute deviation from the mean (i.e., the L_1 norm, see sec. 2.1.1, p. 14), are even more susceptible to this problem as they facilitate the solution by assuming normally distributed returns [234, p. 10–11]. How this problem affects the application of portfolio theory to the ASP is discussed in section 2.5.4. The next sections highlight important differences between financial and algorithmic portfolios and briefly survey existing approaches to algorithm portfolio construction.

2.5.2. From Financial to Algorithmic Portfolios

As Constantinides and Malliaris put it, *"Portfolio selection involves making a decision under uncertainty"*[43, p. 1]. Just as financial portfolios consist of carefully chosen assets that shall maximize the overall future performance with minimal risk, it is possible to create simulation algorithm portfolios with similar properties. Instead of coping with the uncertainty of future economic developments, they cope with the uncertainty regarding the kind of simulation problems that are encountered in the future, thereby minimizing the risk of performing badly on them.

However, choosing an algorithm portfolio does *not* solve the ASP as defined in section 2.1: instead of constructing a selection mapping S, portfolio theory merely allows to choose a (weighted) subset $A \subseteq \mathbb{A}$ of algorithms that are *likely* to perform well in the future. Financial portfolio selection therefore allows

[20] In case $i = j$, $\sigma_{i,j}$ represents the variance of R_i. This is required for equation 2.20 to hold.

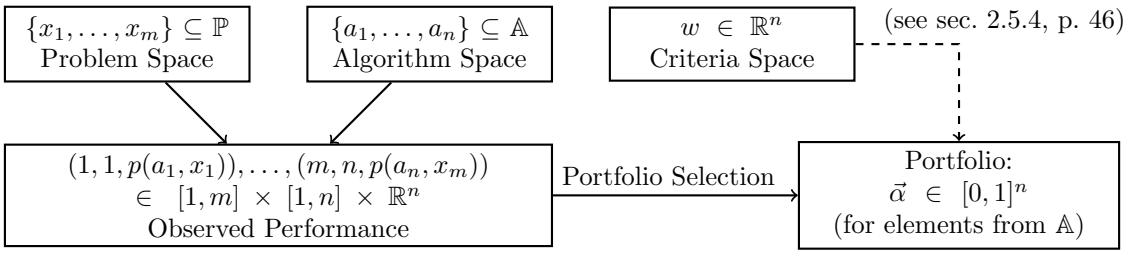

Figure 2.9.: The selection of algorithm portfolios for n performance aspects, i.e., an n-dimensional performance space $\mathbb{R}^n$. It relies on observed performance data that merely points to the indices (m, n) of the corresponding problem and algorithm. No problem features or algorithm properties are considered any further. section 2.5.4 (p. 46) discusses how user criteria can be incorporated into the selection process.

to *simplify* the ASP by applying statistical methods, rather than to solve it. Additionally, classical portfolio selection as outlined in section 2.5.1 does not rely on any problem features, it is restricted to the analysis of past performance. This is an important advantage, as performance can be recorded automatically while the definition and extraction of problem features is likely to require additional programming efforts. Hence, portfolio selection should be easy to automate. Figure 2.9 sketches the main idea of algorithm portfolio selection, and how it relates to the ASP.

Several variants of the portfolio selection problem have been analyzed over the last decades [43]. Many allow short-selling, i.e., associating assets with negative weights so that the constraints $\alpha_i \in [0, 1]$ can be relaxed. However, this relaxation is not meaningful for algorithm portfolios: an algorithm cannot be selected *less than* never. Another common assumption is the existence of a risk-less asset [286], such as government obligations. Again, this may be sensible in the financial domain, but cannot be transferred to algorithms easily: for example, a risk-less simulation algorithm with respect to execution speed would always take the same amount of time to solve a simulation problem, regardless of the problem's structure. Such algorithms are rare at best. Finally, stock performance can also be viewed as a time series (e.g., [210]). This might not be a valid assumption for algorithms either, e.g., in case the performance data for portfolio construction is gathered from a sequence of benchmark models that were executed in arbitrary order. Note that the selection problem defined in section 2.5.1 does *not* make this assumption, i.e., temporal trends in the sequence sampled from the R_i are not considered. Instead, the method considers the correlation of samples at equal points in time. All these aspects make it hard to directly translate findings and methods from financial theory to algorithm portfolios, which may explain why little research on algorithm portfolios is based on financial theory.

Since financial portfolios are inherently applied in parallel, i.e., the capital is allocated to assets which then yield return concurrently, first approaches to algorithm portfolios have followed a similar approach. In [149], Huberman et al. investigate a portfolio approach for graph-coloring with two independently running instances of a *Las Vegas algorithm*, i.e., an algorithm that always solves the problem correctly but contains some stochastic element. Each instance receives a share $\alpha_i \in [0, 1]$ of CPU time, so that $\alpha_1 + \alpha_2 = 1$. Given the random variables T_1 and T_2 that represent the time both instances need to solve the problem, respectively, the overall portfolio runtime can be written as a random variable T_p with [149, p. 51]:

$$T_p = \min\left(\frac{T_1}{\alpha_1}, \frac{T_2}{\alpha_2}\right) \tag{2.22}$$

Given that the discrete probability distributions of T_1 and T_2 are characterized by the probability mass functions $\rho_{T_1}(t)$ and $\rho_{T_2}(t)$, the probability mass function $\rho_{T_p}(t)$ of the portfolio can be written as [149, p. 51]:

$$\rho_{T_p}(t) = \left(\sum_{t' \geq \alpha_1 \cdot t} \rho_{T_1}(t') \right) \cdot \left(\sum_{t' \geq \alpha_2 \cdot t} \rho_{T_2}(t') \right) - \left(\sum_{t' > \alpha_1 \cdot t} \rho_{T_1}(t') \right) \cdot \left(\sum_{t' > \alpha_2 \cdot t} \rho_{T_2}(t') \right) \tag{2.23}$$

The first term on the right side denotes the probability that both instances are finished at some time greater or equal to t, while the second term denotes the probability that both instances are finished at some time greater and *unequal* to t. In other words, the subtraction removes all outcomes except those where at least one of the instances solves the problem at time t and the other instance has not already solved the problem before time t. Note that all times in equation 2.23 are scaled by α_1 and α_2 respectively, as the instances have to share the CPU. By changing α_1 (and thereby α_2, since $\alpha_2 = 1 - \alpha_1$) the probability mass function ρ_{T_p} of the portfolio runtime can now be adapted. Since the probability mass functions ρ_{T_1} and ρ_{T_2} are identical in this case, i.e., they both refer to the same algorithm, it suffices to vary α_1 from 1 to $\frac{1}{2}$. Plotting mean and variance of the portfolio runtime yields results similar to figure 2.8 [149], i.e., the efficient frontier of the portfolio set can be determined. This approach differs from the classical portfolio selection technique (see sec. 2.5.1) in that it does not consider any correlations between algorithms on differing problem instances, but instead approximates the probability distributions of a performance measurement on a single problem instance. A similar approach that constructs algorithmic portfolios via empirically estimated probability distributions is presented in [113, 114]. It is also applied to computationally hard constraint satisfaction problems (see sec. 2.7, p. 56).

2.5.3. Algorithm Portfolio Variants

The approach described in section 2.5.2 assigns a fixed share of CPU power to each algorithm, and both of them have to be executed in parallel. These restrictions have been relaxed in more recent approaches.

Parallel and Interleaved Portfolios

In [114], Gomes and Selman consider three types of portfolio execution: run all algorithms in *parallel*, let the algorithms run *interleaved* on a single resource, or use a restart strategy, i.e., determine when to stop and re-start an algorithm. This only makes sense when working with stochastic algorithms and is rather loosely related to the other two modes of operation. Restarting is predominantly used in the context of heuristic search methods.

The difference between parallel and interleaved algorithm portfolio execution is more on a technical level: while the theoretical formulation of the problem allows to assign any real value $\alpha_i \in [0, 1]$ to weight the importance of an algorithm, it is not trivial to realize a truly parallel execution with corresponding shares of computing power assigned to each algorithm. Furthermore, the management of the parallel threads may also introduce some overhead, which is usually not considered in portfolio construction approaches. This overhead can be limited by constraining portfolio size, as discussed in section 2.5.1. The results in [114] suggest that the size of a portfolio and the way in which the algorithms are actually executed, i.e., truly parallel or interleaved, may influence the shape of the efficient frontier and the overall portfolio performance. However, it is not yet clear to what extent these results are dependent on the specific application domain, i.e., combinatorial search.

Dynamic Portfolios

Other kinds of algorithm portfolios do not rely on fixed weights, but attempt to learn them during execution. Gagliolo and Schmidhuber call such algorithm portfolios *dynamic*, in contrast to *static*

ones with fixed weights [96]. They motivate their approach by highlighting the implicit assumptions of a prior decision on algorithm weights: the training set of past problems needs to be representative of future problems (see problem of inductive knowledge, sec. 1.4, p. 8), and an accurate prediction of algorithm run time needs to be possible for a given problem instance — i.e., the task needs to be *learnable* (in the sense of sec. 2.3.3). Furthermore, they claim that the training cost of executing all algorithms on the training set as well as generating and evaluating prediction functions is often neglected and can be considerable [96, p. 296].[21] In [93, 94, 95, 97], Gagliolo et al. develop the *AOTA framework*, which stands for Adaptive Online Time Allocation, to overcome these problems with dynamic parallel algorithm portfolios.

Given a sequence of problems $p_1, \ldots, p_m \in \mathbb{P}$ and a set $\mathbb{A} = \{a_1, \ldots, a_n\}$ of applicable algorithms, an AOTA mechanism repeatedly decides upon the algorithmic weight $\alpha_i \in [0, 1]$ for each algorithm, so that $\sum_{i=1}^{n} \alpha_i = 1$. Similar to the approach described in section 2.5.2, the algorithm weights represent the share of CPU time to be invested in each algorithmic asset. Consequently, an algorithm a_i is executed for $\alpha_i \cdot \Delta T$ CPU time, where ΔT is called *time slice* and denotes the amount of time that passes until the next invocation of the mechanism. The AOTA framework therefore considers parallel portfolios as defined above.

The weights $\alpha_1, \ldots, \alpha_n$ are calculated by two functions, f_τ and f_x:[22] f_τ predicts the time τ_i that algorithm a_i still needs to solve the current problem, and f_x updates the currently selected portfolio, i.e., the current algorithm weights, depending on the predicted τ_i. The prediction is based on a *history* H:

$$H = \{H_1, \ldots H_n\}, H_i = \{(d_i^r, t_i^r) | r = 0, \ldots, h_i\}$$

with r being the index for past rounds of reallocation, t_i^r is the time allocated to algorithm a_i at round r, and d_i^r is a vector containing features of the current problem, algorithm features (e.g., its parameters), and also some information on the current state of the algorithm a_i. Galgliolo and Schmidhuber illustrate the approach by applying it to genetic algorithms [97]. Here, d_i^r could contain the mutation rate as an algorithm's parameter, and the current state of the algorithm could be reflected by the average fitness of the current population.

Gagliolo et al. distinguish between *intra-problem* and *inter-problem* AOTA. Intra-problem AOTA considers only results from the current problem instance, so that the execution times of the algorithms do not need to be fed back for improving f_τ, which is fixed. This behavior is called *oblivious* by the authors, as it discards the knowledge gained during execution when faced with the next problem instance. In [97], f_τ is derived by linear regression for intra-problem AOTA. In contrast, inter-problem AOTA provides 'life-long learning' [94] in that f_τ itself is improved after finishing problem p_i, so that solving p_{i+1} is done with a prediction function f_τ that was built by considering all prior performances of the algorithms on instances $p_1, \ldots, p_i$. Inter-problem AOTA was evaluated with neural networks [93] and Bayesian approaches [95] for f_τ.

In any case, the τ_i predicted by f_τ are finally passed over to f_x, which now translates them to algorithm weights α_i. The function f_x can also take many forms, e.g., it may rank the algorithms with respect to the predicted solution time and assign shares of the form 2^{-r_i}, r_i being the rank of algorithm a_i: the fastest algorithm receives half of the processor time, the second best a quarter, and so on. For inter-problem AOTA, this form of f_x might lead to problems, because early prediction mistakes could lead to misinterpretations: for example, the algorithm wrongly assumed to be the best could receive twice the CPU time of a potentially superior algorithm, which would go unnoticed as the portfolio stops when the first algorithm finds a solution. This problem can be alleviated by gradually moving from $\alpha_i = \frac{1}{n}$, i.e., a uniform parallel portfolio, to $\alpha_i = 2^{-r_i}$ [96, p. 5]. The underlying problem here is again the trade-off between exploration and exploitation (see sec. 2.3.2, p. 29).

This similarity to reinforcement learning is exploited in [96], where Gagliolo and Schmidhuber extend the inter-problem AOTA mechanism by replacing f_τ and f_x with a set of *time allocators*

[21] This is the metareasoning-partition problem, as described in sec. 2.3.3, p. 33.
[22] Not to be confused with problem features in the context of the ASP, sec. 2.1, p. 13.

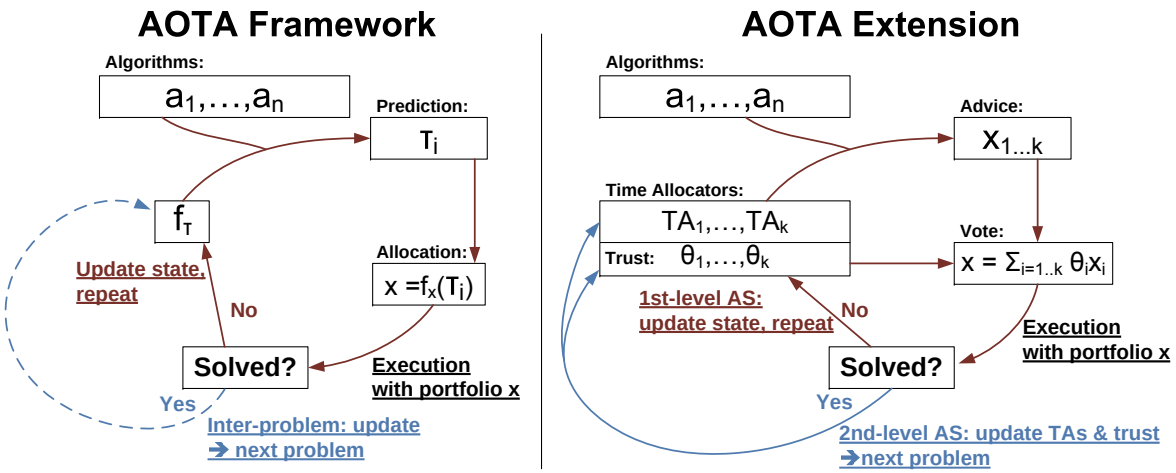

Figure 2.10.: The AOTA framework for parallel algorithm portfolios. The left side shows the framework as described in [93, 94, 95, 97]. The brown arrows and labels denote the default feedback loop when running on a single instance, whereas the dashed blue line illustrates the knowledge transfer to f_τ in the non-oblivious case, i.e., inter-problem AOTA. The right side shows its structurally similar extension presented in [96], which builds upon a multi-armed-bandit policy from [10] to add a second level of algorithm selection (blue): apart from adjusting the time allocation during the processing of a problem instance (brown), the method continuously adjusts the trust it has in the time allocation schemes, i.e., algorithm selectors, that are applied to the problem.

$TA_1, \ldots, TA_k$. Each time allocator operates as a combination of f_τ and f_x in the inter-problem AOTA framework, i.e., it selects a portfolio based on the history H, and may incrementally improve its internal model. Employing a whole *set* of time allocators to solve the ASP requires an additional decision-making procedure on top, as sketched in figure 2.10. This procedure then has to solve *another* selection problem, i.e., it has to select the best time allocation algorithm from the set. In this context, the ASP is also known as *meta-learning*: each of the time allocators is learning to select the best algorithm, while the procedure on top learns which of the learning allocators performs best — and therefore performs meta-learning (see sec. 2.7, p. 55).

In [96], the meta-learning is done by a policy from [10] which plays the full-information multi-armed bandit problem: each TA_i proposes a certain portfolio X_i, which can be regarded as the advice by an expert. As depicted in figure 2.10, the advice is then joined to a single portfolio via a 'voting' scheme that weights experts by the trust θ_i that the meta-learner puts in them. After a problem instance has been solved, the policy distributes the received reward over the trust values θ_i, depending on how much each TA_i favored the winning algorithm in its portfolio proposal X_i. By this, the policy learns which experts are likely to select the winning algorithm. To avoid bad overall performance in case none of the TA_i is able to learn portfolio selection properly, a uniform time allocator that allocates the same share of CPU time to each algorithm is included. This allows the meta-learner to detect when the custom time allocators learned enough to outperform a static uniform portfolio selection.

The time allocators use nearest-neighbor learning, i.e., prediction by considering the most similar past problem instances, in combination with findings from *algorithmic survival analysis*. The latter is based on statistical survival analysis methods from the social sciences, where researchers struggle with sample populations from which individuals might be *censored* for unknown reasons, i.e., they are not considered any longer and it is unknown when the critical event (e.g., death) occurred. Handling such censored samples wrongly may introduce bias, so statistical methods are required that take censoring into account. The same holds for algorithms in parallel portfolios: portfolio execution is stopped when the first algorithm stops, so all other algorithms are censored. It is unknown if and when the critical event, i.e., finding a solution, would have occurred for them. In *algorithmic* survival analysis, the critical event of finding a solution does not necessarily have to occur at all, e.g., for semi-decidable

problems, whereas classical survival analysis assumes that the critical event, e.g., death, eventually occurs with a probability of 1. Statistical methods to account for censored data in a meaningful way may drastically reduce the training cost by *not* letting all algorithms finish, and are especially useful in challenging setups like the AOTA framework, with incremental learning on two distinct levels.

The selection of parallel dynamic algorithm portfolios can also be formulated as a Markov decision process (see sec. 2.3.2), so that an optimal policy for the MDP is also optimal for dynamic selection of a parallel portfolio [256]. In [188, 189], Lagoudakis et al. present another kind of dynamic algorithm portfolio, which is also based on the notion of MDPs but selects a single-asset portfolio *sequentially*: it considers recursive algorithms and selects the algorithm to solve the current sub-problem each time a recursion occurs. It is a dynamic portfolio in that the algorithm choice varies during the execution, but unlike the other presented approaches it lets all selected algorithms work *together*. Reinforcement learning is used to decide which algorithm to choose in which situation.

2.5.4. Portfolios for Simulation Algorithm Selection

While the algorithm selection problem and machine learning are interconnected by approximation theory (see sec. 2.1.3, p. 21, and sec. 2.3, p. 24), the relation between algorithm portfolios and simulation algorithm selection is not that evident. As depicted in figure 2.9, common approaches to algorithm portfolios do not necessarily take into account the same entities as the ASP prescribes, nor do they yield the same results. Furthermore, algorithm portfolios have mostly been applied to a very specific class of problems so far, which has little overlap with simulation problems (see sec. 2.7, p. 56).

Disambiguation: Portfolio Selection vs. Algorithm Selection

The variety of portfolio approaches presented in section 2.5.3, ranging from static to dynamic portfolios and from parallel to sequential ones, brings up the question of what distinguishes a portfolio approach from other algorithm selection methods. It is even suggested that there is no such difference, e.g., in [205, p. 34]: *"In contrast, the empirical-hardness-model-based portfolios discussed above are 1-of-n portfolios. (Since only one algorithm is selected, the sequential/parallel distinction is moot in this case.)"*. This would make the ASP as defined in section 2.1 a special case of algorithm portfolios, i.e., 1-of-n portfolios. Such a redefinition seems misleading, as algorithm portfolios are usually regarded as a *subset* of algorithm selection methods [205, p. 34]: *"The term has since also been used to describe any strategy that combines multiple, independent algorithms to solve a single problem instance"*. It seems natural to follow the latter variant and define an algorithm portfolio approach as any algorithm selection method that selects *multiple* algorithms to solve a *single* problem instance. Nevertheless, portfolio techniques could be used to *simplify* the selection of a single algorithm by reducing the choice and thereby potentially alleviate the metareasoning-partition problem (see sec. 2.4).

Portfolios of Simulation Algorithms

Algorithm portfolio construction is an interesting technique to select subsets of algorithms. It is usually independent of problem features and is often based on purely statistical considerations. However, it is still unclear how well these approaches can be applied to simulation algorithms that are usually not implemented recursively, as required in [188, 189], and are also not solving hard problems in the sense of theoretical computer science, as presumed in [114, 149, 205, 243, 256]. Simulation algorithms are far less complex, and their efficiency requirements usually stem from large problem sizes. Considerable benefits may stem from comparably small optimizations. Besides that, portfolio methods might not cope well with really large sets of algorithms; typically only portfolios with less then 10 distinct algorithms are considered, e.g., in [96, 114, 149, 205, 256]. A notable exception is the inter-problem AOTA framework, which was tested with 76 different setups of a genetic algorithm [95].

Two further issues arise when trying to construct simulation algorithm portfolios. Firstly, some of the selection methods from finance make assumptions regarding the probability distributions of the returns (see sec. 2.5.1), in order to speed up the identification of the efficient frontier. This makes

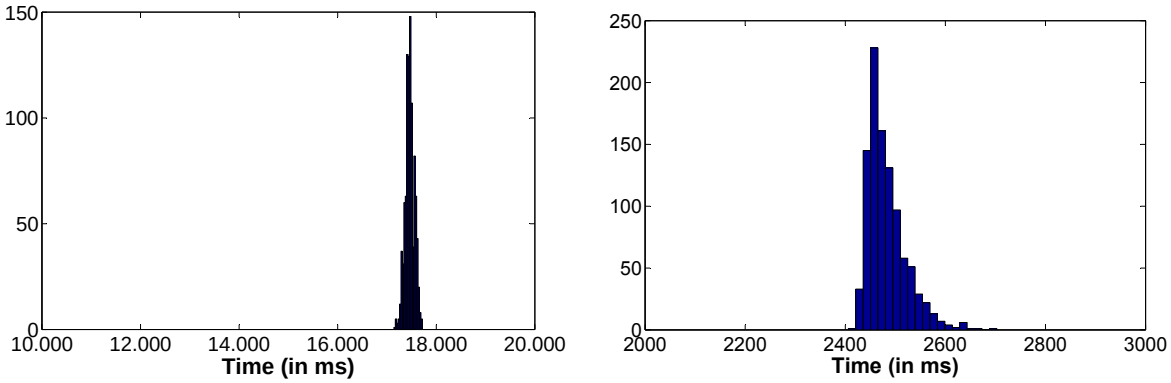

Figure 2.11.: Two sample histograms of simulation algorithm runtime distributions in JAMES II. They are *not necessarily* heavy-tailed, as those considered in [114], but also not necessarily normally distributed either. Left: Execution times of 1000 flat-sequential PDEVS simulator runs on a 100×100 ForestFire benchmark model (see [134]). Right: Execution times of 1000 NRM-B simulator runs on a *CCS* benchmark model (see [158] or p. 149).

it necessary to investigate the *runtime distributions* of simulation algorithms, as execution time is an important performance metric. Secondly, portfolio theory inherently regards a *single* one-dimensional performance metric, i.e., the return of an asset. This has to be reconciled with the ASP's notion of user criteria for multiple performance facets.

Performance Distributions Simulation algorithms typically iterate many times over a loop that lets them proceed in simulation time. The runtime of each loop cycle is influenced by many factors, e.g., the current state of the model, but also the current state of the hardware resources as well as external load from other programs. Many of these factors can assumed to be independent of each other, e.g., model state and external load, so one might be tempted to retreat to the central limit theorem [277, p. 70–71] and assume simulation algorithm run times to be always normally distributed. As figure 2.11 illustrates, this is not necessarily the case: the runtime distribution on the right is clearly skewed, with a much shorter head than tail. The reason for this is that *some* unobserved random variables which impact the runtime are neither identically distributed nor independent, as the central limit theorem presumes. For example, the model's computational load within a loop cycle depends on former model states, and hence the model's former load. Therefore, the shape of the runtime distributions cannot be generalized over all simulation algorithms. The left plot in figure 2.11 shows another empirical distribution, with very different properties: it is very narrow, i.e., there are so few outliers that the runtime can be predicted easily. Just as in economics, the shape of the runtime distribution can be an important measure of predictability [300], and hence needs to be considered carefully. Similar investigations are also necessary for any other performance criteria before a portfolio selection technique with strong implicit assumptions on return distributions can be applied. This problem appears in many application domains of algorithm selection and does *also* depend on the hardware, not only the model — even when considering relatively straightforward algorithms. For example, Vuduc et al. observe a *"relatively long tail"* [319, p. 77] of runtime distributions for different implementations of matrix multiplication, depending on the hardware architecture.

Portfolios for Multiple User Criteria The most straightforward way to extend portfolio selection to multiple user criteria, i.e., different aspects of algorithm performance, would be to consider $k \cdot n$ -dimensional performance vectors for the k algorithms and n performance facets, i.e., vectors of the form

$$(p_{1,1}, \ldots, p_{1,n}, p_{2,1}, \ldots, p_{k,n})$$

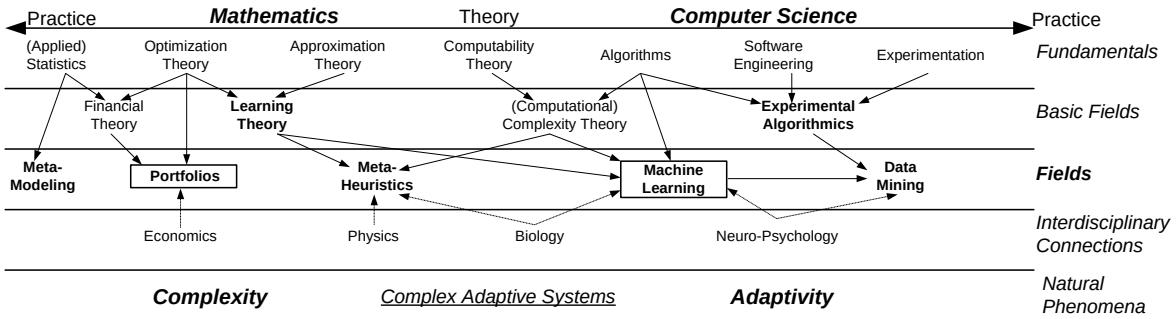

Figure 2.12.: Relevant scientific interconnections among the various disciplines that can be used to solve the algorithm selection problem. Bold type highlights the most relevant disciplines, additional boxes mark those that gave rise to specific solutions. Related fields are placed near to each other, and fields that contribute more theoretically than practically to solving the ASP are placed more centered.

so that all performance measures $p_{i,1}, \ldots, p_{i,n}$ that describe algorithm a_i's performance on a problem are grouped together. The user criteria $w \in \mathbb{R}^n$ could then be used to construct k-dimensional vectors that represent the weighted performance $||g(p(a_i, x_i), w)||$ for all algorithms $a_i \in \mathbb{A}$ and problems $x_i \in \mathbb{P}$ (see sec. 2.1). The performance weighting by the user criteria, however, merely allows to construct portfolios for *specific* $w \in \mathbb{R}^n$. Thus, selection cannot be conducted in advance, i.e., before the user has entered the criteria. It may take considerable time, so that usability is hampered.

Several simple strategies may help to overcome this issue, but they add complexity and do not guarantee a portfolio that is truly efficient. For example, a small number of user criteria can be tackled by filling a look-up table with different weights distributions, and then use the efficient frontier that belongs to the most similar weights from the table when confronted with new user criteria. Finally, one could calculate the efficient frontier for every single metric in isolation, and then combine the resulting portfolios in some manner.

Note that the number of user criteria for portfolios is always even, as not only the importance of the performance aspect, but also the user's *risk-aversion* regarding that aspect, can be taken into account. The risk aversion λ is used to select the actual portfolio from the efficient frontier (see sec. 2.5, eq. 2.21, p. 40). This would, for example, allow users to take high risks for good execution speed while being very conservative when it comes to accuracy. The explicit consideration of risk as a first-class metric that can be controlled (to some extent) is one of the most appealing features of algorithm portfolios.

2.6. Categorization of Algorithm Selection Methods

The past sections have formally introduced the algorithm selection problem and described several potential approaches to solve it (or at least some of its aspects). These approaches originate from various disciplines: theoretical computer science (sec. 2.2), artificial intelligence (sec. 2.3), software development (sec. 2.4.3), and economics (sec. 2.5). An overview of the relations among the most relevant areas is sketched in figure 2.12; however it should not be considered complete. As Smith-Miles points out, *"It is surprising how little intersection there has been in the relevant developments in these different communities, and how the vocabulary has evolved in different directions, making searching for relevant papers in other disciplines more difficult"* [292, p. 21].

Since good overviews and categorizations, such as [96, 292], are rare and survey the field from the perspective of their specific community, it is useful to develop a more general categorization of algorithm selection methods, based on the same vocabulary that describes the ASP itself. This helps to discuss related work, to identify similar methods, and to appreciate the virtues of differing viewpoints.

2.6.1. Categorization Aspects

Most approaches to algorithm selection do not consider feature selection or treat it as a separate problem, so that the following categorization focuses on approaches that explicitly or implicitly aim at solving the best selection mapping problem (def. 2.1.2, p. 15). The categorization is based on four basic aspects of algorithm selection methods, which can be phrased as questions and may comprise several sub-aspects:

- **Problem**: Which specific ASP is solved?
- **Data**: What kind of data is analyzed to do so?
- **Algorithms**: Which algorithms are eligible for selection?
- **Solution**: How is the ASP solved?

Problem

Problem-Type Firstly, one can distinguish between *decision* and *optimization* problems [96, 205]. The close relationship between optimization and the ASP has already been discussed in section 2.1.3 (p. 20). Phrased as a decision problem, one could ask whether an algorithm $a \in \mathbb{A}$ reaches a certain performance level $p_l = ||p(a, x)||$ for a problem $x \in \mathbb{P}$. Decision problems are often considered in theory, e.g., Guo bases its undecidability proof for the ASP on a similar reformulation (see sec. 2.1.3). Another example are algorithm portfolios, e.g., when they are constructed by investigating the probability that a certain algorithm reaches a certain level of performance [114]. Note that the above discussion relates to the ASP problem as such, *not* to the kind of problems for which algorithms shall be selected (this distinction is made as well, e.g., in [96, 205]). Most of the aforementioned methods are based on the optimization viewpoint, e.g., the MABP policies described in section 2.3.2 (p. 29), so this perspective seems more natural.

Problem- and Selection-Granularity Another distinction endorsed in [96] is that between solving the ASP *per-set* or *per-instance*.[23] In case of simulation problems, a per-instance solution is applicable even if only one simulation problem instance shall be solved (see sec. 2.4.1, p. 34), i.e., a single simulation run shall be executed. This can be realized, for example, by letting a previously learned selection mapping decide on a suitable algorithm. In contrast, MABP policies (see sec. 2.3.2, p. 32) need to consider several problem instances (i.e., replications) before they become effective: they solve the ASP per-set.

The same distinction can be made for approaches that find a selection mapping as defined in section 2.1, and those that select a *(sub-)set* of suitable algorithms for a given problem. In contrast to *single-selection* methods, these *set-selection* approaches reduce the ASP hardness rather than solving the ASP. Many algorithm portfolio approaches (see sec. 2.5, p. 38) fall into this category, e.g., [95, 114].

Data

Theoretical vs. Empirical If the best selection mapping $S \in \mathbb{S}$ is solely chosen on theoretical grounds, e.g., by analyzing the time and space complexity of all algorithms in $\mathbb{A}$ (see sec. 2.2), this selection process is called *theoretical*. It subsumes rigorous mathematical deductions of the best algorithm for some problem features and user criteria, but also intuitive decisions, e.g., as mentioned in [81, 257], as long as these are not based on any experimental data that has been recorded beforehand. Intuitive algorithm selection often implicitly steers algorithm design, e.g., when a developer decides to implement a specific synchronization scheme for a parallel and distributed simulation engine.[24]

[23] A similar distinction between single- and multiple-instance problems is made in [145].

[24] Though *experienced* developers will make this decision by considering their past experience, i.e., some kind of experimental data, in which case their selection is empirical.

Thereby, theoretical algorithm selection reduces the set $\mathbb{A}$ of all algorithms that could solve the problems in $\mathbb{P}$ to a usually rather small subset $\mathbb{A}_0 \subset \mathbb{A}$ of algorithms that are actually implemented. Besides $\mathbb{P}$ itself, such an analysis may also consider the feature space $\mathbb{F}$, as well as some concrete sample problems $x_1, \ldots, x_n \in \mathbb{P}$ and user criteria. However, it must not consider any performance data, i.e., evaluations of $p(a, x)$ (see def. 2.1.1, p. 14). If a method does rely on evaluations of $p(a, x)$, it is called *empirical*. Note that this definition of empirical algorithm selection does *not* imply that empirical methods forgo theoretical considerations; they just *also* incorporate experimental data in the form of $p(a, x)$ evaluations. Since it is not very promising to automate theoretical algorithm selection (see sec. 2.2), the focus of the categorization is on empirical methods that may or may not rely on additional theory (see discussion in [142, 235]). In [117, p. 62] a similar distinction between 'experimental approaches' and 'analytical approaches' is made, the latter being restricted to *"use purely mathematical methods"*.

Data Collection If the ASP is solved empirically, it is also important to distinguish between different sources of empirical data. For example, the approach in [25] collects data from profiling the algorithms with a calibration toolkit on the target platform. This process is done during the installation of the system and is not repeated. Such collected data are therefore *platform-dependent*, i.e., the data stems from the same kind of machine on which the selection shall take place. A similar mechanism that also relies on platform-dependent data is presented by Yu et al. in [338]. The collected data are often also *synthetic*, i.e., they are recorded by examining artificial workloads. Finally, the data can be regarded as a *snapshot* in this case, i.e., data collection is only done once, to construct a good selection mapping.

In contrast, *platform-independent* experimental data does not depend on the execution platform. For example, data on the accuracy of simulation algorithms does usually not depend on the underlying hardware.[25] It could therefore be shared among all users. On the other hand, some data are not only *platform-* but rather *machine-dependent* — e.g., when GPU-based simulation algorithms are used, the specific GPU model may have a strong impact on overall performance (e.g., [280]). The difference between machine- and platform-dependent data hence lies in their general applicability, i.e., the extent to which specific hardware features are exploited.

In [338], synthetic experiments are conducted to fit the parameters of a selection mapping. If some data is extracted from real-world problems instead of synthetic ones, this data is called *realistic*. Considering synthetic problems instead of real ones is often advantageous for performance experiments (see sec. 7.3.1, p. 145), but using real problems ensures that the collected data is (to some extent) representative. Instead of only considering data from a single snapshot, i.e., which was recorded *once*, some methods also allow to reconstruct the selection mapping based on new empirical results. In this case, data collection is said to be *continuous*. For example, approaches from reinforcement learning rely on continuous data collection in the form of reward (see sec. 2.3.2). Another possibility would be to let the user *trigger* collection manually. Obviously, many kinds of data, e.g., synthetic and realistic data, can be combined to form a sufficiently large data base. Practical implementations may therefore rely on data that fall into several of the above categories.

Algorithms

Another important aspect is the structure and the size of the algorithm set $\mathbb{A}$, which is the co-domain of the selection mapping to be found. For any implementation on a physical machine, $\mathbb{A}$ clearly has to be finite. However, some methods are prohibitive for very large sets of algorithms, e.g., MABP policies may converge quite slowly when faced with a high number of options. Large algorithm sets may not even be enumerated, but rather defined implicitly by the specific techniques to *explore* $\mathbb{A}$. For example, a selection method could rely on techniques from *genetic programming* [279, p. 133] to

[25] That is, if a high-level language is used and the code has been developed carefully (i.e., the same truncation errors occur on all platforms, and so on).

generate a huge variety of eligible algorithms. If such techniques are used to explore $\mathbb{A}$ automatically, the algorithm set is said to be *generated*, e.g., as discussed in [319].

On the other hand, if $\mathbb{A}$ is simply a set of distinct algorithms $a_1, \ldots, a_n$, each of which represents an alternative implementation to solve a problem, the algorithm set is called *monolithic*, i.e., it consists of monolithic algorithms. Such algorithm sets are often considered for algorithm portfolios, e.g., in [149, 203]. Two other kinds of algorithm sets fall in-between these two extremes.

Firstly, many algorithms may be monolithic but can still be parameterized. This often makes the algorithm space $\mathbb{A}$ rather large in theory, but the situation is still quite different from the case of generated algorithms in $\mathbb{A}$: here, it often seems reasonable to assume that algorithms $a_i, \ldots, a_j$, which all rely on the same implementation and only vary by parameterization, will still have rather similar overall behavior. This allows to consider these parameters in a joint performance prediction function $\widehat{perf}_{a_{i,\ldots,j}}$ (see eq. 2.13), and thus makes the problem considerably easier than an equally large set of truly distinct algorithms. An example for such *monolithic-parametric* algorithm sets is presented in [25].

Secondly, the algorithms in $\mathbb{A}$ might not be monolithic black boxes, but rather *combinations* of exchangeable algorithms that solve certain sub-tasks, e.g., as in [146]. Such viewpoint is also likely in component-based systems with a focus on reusability and flexibility, such as JAMES II. Both the actual algorithms and their sub-algorithms may be parameterizable. Due to a combinatorial explosion, the algorithm set $\mathbb{A}$ might become very large, even though all included algorithms have been implemented manually. Nevertheless, the structural information inherent to the algorithm combinations might be useful to steer algorithm selection and could therefore simplify the problem. Such algorithm sets are called *combinatorial-parametric*.

Solution

Solution and Application Time Two major aspects of any ASP solution are at which point in the life cycle of a software system the ASP is *solved*, and at which point the selection mapping S, which represents the solution, is actually *applied* to select an algorithm. Four main times should be distinguished:

1. **Design Time**: The time span in which a software is developed.
   →*For example, when a new simulation algorithm X is implemented.*

2. **Compile Time**: The time span in which executable files are generated for a release.
   →*For example, when the implementation of X is compiled.*

3. **Run Time**: The time span in which the software as such is executed.
   →*For example, when the simulation system that uses the implementation of X is started.*

4. **Processing Time**: The time span in which a concrete problem is solved by the software.
   →*For example, when the implementation of X is applied to a simulation problem.*

Both solution and application time of an ASP can be any of these times, with the obvious restriction that the solution time has to precede (or equal) the application time. The basic ASP definition from section 2.1 makes no assumption on the solution time, i.e., the time at which a suitable selection mapping is found. However, the resulting selection mapping S depends on the features of the specific problem, so that the application of a 'true' ASP solution has to happen at processing time — the only time at which all problem features can be known. Contrarily, theoretical algorithm selection usually implies that both solution and application happen at design time, since theoretical analysis of algorithms is hard to automate (see sec. 2.2, p. 22). Similar differentiations by time can be found in related categorizations, e.g., in the taxonomy for self-adaptive software systems presented in [230]. As another example, Vuduc et al. distinguish between algorithm selection at compile time and run time [318] (and combinations thereof [319]). However, what they regard as the run time is referred to as *processing time* in the above enumeration, since their algorithm selection rules rely on the features

of the specific problem instance. The run time in the above sense would be solution and application time of an ASP method if, for example, this method is executed by users (i.e., after compilation) and the results are used to determine a fixed algorithm that is selected in the following. This selection is then made *before* any specific problem instance is processed. Many simulation users are implicitly conducting this kind of algorithm selection, e.g., when they work with simulation software like MATLAB [223] and change their default settings to the numerical integrator they deem best because of past results.

Solution and Application Frequency Apart from the time at which the ASP is solved and its solution is applied, it can also be interesting to consider the *frequency* with which this is done for solving a *single* problem. Most ASPs are solved once and the selection mapping is then applied frequently — otherwise the efforts of searching for a good selection mapping could not be amortized. Nevertheless, the selection mapping is usually considered only one time per problem. Such methods have a solution and application frequency of *once*. Other approaches, like the dynamic algorithm portfolios introduced by Gagliolo and Schmidhuber [96] (see sec. 2.5), *continuously* update and apply the selection mapping at processing time; their frequencies are therefore called *continuous*. Alternatively, both the ASP solution and its application could be triggered from time to time, e.g., by the users of the system. This kind of frequency will therefore be called *triggered*, whereas a *continuous* frequency implies automation.

Solution Procedure Methods to find suitable selection mappings have some general properties that help categorization. Since most techniques are in some sense related to the field of machine learning, or have even been developed in this context [292], it seems suitable to apply some of its rich terminology to the given problem; similar reasonings can be found in [96, 319].

At first, one should distinguish between selection mappings that *classify* a problem with respect to the most suitable algorithm and those that *predict* the performance of all algorithms for the problem and then choose the algorithm that is predicted to be best (see sec. 2.3, p. 24). A similar distinction is put forward by Leyton-Brown et al., who argue in favor of prediction approaches [205, p. 36–37].

Moreover, some methods operate in *batch* mode, i.e., they consider all available data at once for searching a good selection mapping, whereas others are *incremental*, i.e., they can improve an existing selection mapping on the basis of new data. A selection technique may also be oblivious, i.e., it may entirely dismiss data gathered from old problems. This term has been taken from [96], where oblivious methods are used to implement dynamic algorithm portfolios. A non-oblivious batch method would therefore consider both past and current data at once to search a new selection mapping from scratch, whereas a non-oblivious incremental method would also take into account the current selection mapping, usually as a substitute for raw past data. Their oblivious counterparts would be restricted to data on the current problem.

In this sense, non-oblivious incremental methods seem most suitable to achieve the ultimate goal of 'life-long learning' [94], i.e., to have a mechanism that continuously solves the ASP and in some sense considers all data from the past. Unfortunately, such methods are hard to come by.

2.6.2. Summary

A visual summary of the categorization is given in figure 2.13. Several other papers, e.g., [60, 96, 292], provide simpler yet less comprehensive categorizations, mostly focusing on the principal solution techniques. These can, however, be mapped easily to this categorization. For example, Dongarra and Eijkhout [60, p. 126–127] discriminate between four operation modes of self-adaptive numerical software:

1. **Off-line optimization**: Solution and application happen *once* at *compile time*.
2. **Hybrid off-line/run-time optimization**: A solution is still found at compile time, but its application is done *once* at *processing time*.

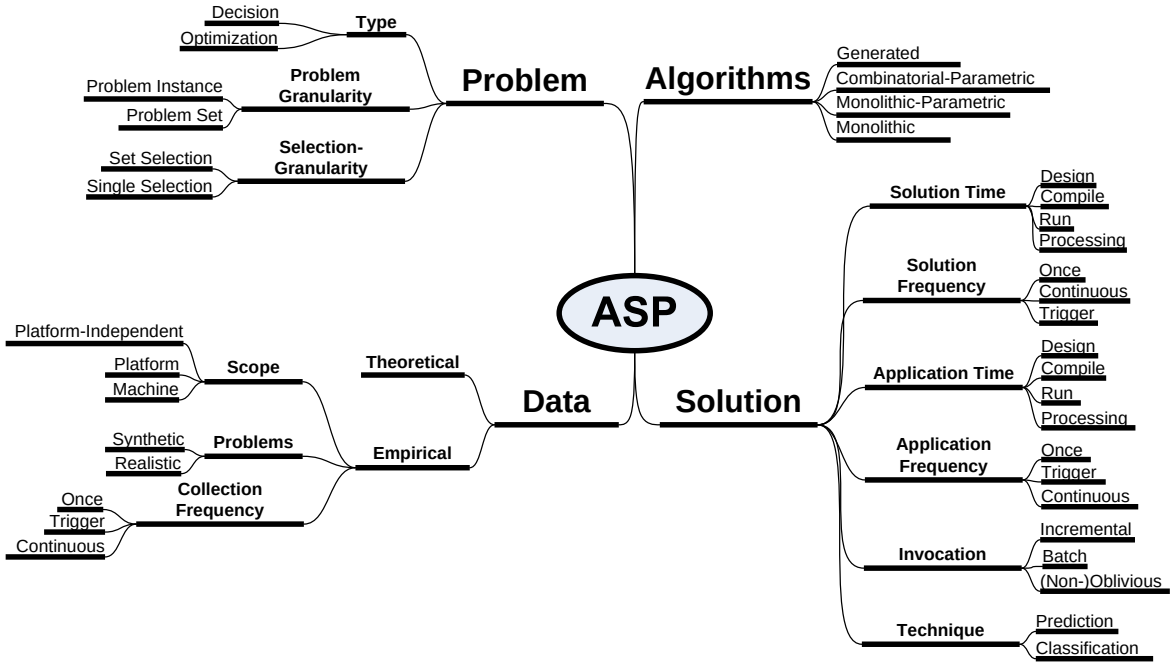

Figure 2.13.: General properties of algorithm selection methods as discussed in section 2.6.1. Some approaches apply to multiple categories.

3. **Completely run-time optimization**: Both solution and application happen *once* at *run time*.

4. **Feedback directed optimization**: This category represents methods that include *"running the program, collecting profile and other information [...] and recompiling with this information, or saving it for future reference when similar problems are to be solved"* [60, p. 127]. In terms of the proposed categorization, this category therefore contains all methods that rely on empirical data and solve the ASP at either compile or processing time. The authors concede that this scenario is not disjoint to 'completely run-time optimization'; in fact it overlaps with any of the other three categories.

The categorization from [60] does neither regard different sub-problems (e.g., set-selection vs. single-selection) nor the structure of the algorithm set — which might be only natural for the specific application area the authors work in, i.e., empirical tuning (see sec. 2.7, p. 54). As another example, the distinction between *static* and *dynamic* algorithm portfolios made in [96] (see sec. 2.5.3, p. 42) can now be described as applying the ASP solution *once* at the beginning of *processing time* on the one hand, and applying it *continuously* at *processing time* on the other hand. Similarly, the categorization of algorithm portfolios into *sequential*, *partly-sequential*, and *parallel* — as proposed by Leyton-Brown et al. in [205, p. 34] — merely refers to selection granularity: the definition of a sequential portfolio equals an algorithm selection method with single-selection granularity, i.e., only one element from $\mathbb{A}$ is selected. Partly-sequential portfolios, in contrast, solve the ASP with set-selection granularity by selecting a subset $A \subset \mathbb{A}$, and parallel portfolios execute *all* algorithms in parallel, i.e., $A = \mathbb{A}$. Consequently, parallel portfolios can only be regarded as algorithm selection if some form of selection takes place at processing time, since all algorithms are 'selected' at the beginning. For example, this approach is pursued in [96], and is therefore also categorized as *dynamic* by its creators.

These examples illustrate that there is no common nomenclature throughout the domains in which algorithm selection techniques are used. This strongly hampers the discussion and evaluation of new contributions, as well as their transfer to other problem domains [292]. The presented categorization aims at overcoming these problems. For example, the distinctive feature of portfolio approaches, i.e.,

that more than one algorithm is applied to the same problem (see sec. 2.5.4, p. 45), can now be more rigorously defined by demanding either a set-selection *or* a continuous application of the ASP method at processing time.

All in all, the categorization suggests that an algorithm selection method should always be discussed in the context of four main ASP aspects (see fig. 2.13): the actual *problem* type it solves, the set of *algorithms* it selects from, the *data* that is considered, and the *solution* mechanism that is employed. Additionally, one could also consider and compare the size and the dimensions of the sets $\mathbb{A}_0, \mathbb{P}_0, \mathbb{F}$ etc. from the formal ASP definition. This approach has been followed in [292] and allows quantitative comparisons across the fields. However, such categorizations still lack information on the nature of the data, the times at which the ASP is solved, and so on.

2.7. Applications of Algorithm Selection

This section briefly overviews the most important application domains of algorithm selection and tries to clarify the commonalities and differences between the fields, by referring to the categorization introduced in the last section. A tabular overview of the most relevant approaches and some of their properties is given in the appendix (p. 221).

Fundamental Algorithms

In his work on sorting, Knuth complains that *"It was fun to learn all the techniques, but now we must face the horrible prospect of actually deciding which method ought to be used in a given situation."* [181, p. 379]. This assessment seems to be shared by many other computer scientists, as there are relatively many approaches to optimize sorting performance by algorithm selection. The dependence of such a selection on the given hardware architecture is illustrated in [191]. Empirical tuning of sorting algorithms, combined with a simple algorithm selection at processing time, has been presented in [206]. Sorting as such can even be learned by genetic algorithms, as [193] shows in the context of autonomous computing. This can be regarded as a very simple example of algorithm selection on *generated* algorithms, where the selection of the best one is due to evolutionary mechanisms.

Lagoudakis et al. [188, 189] interpret the recursive calls for sorting sub-lists as a Markov decision process and consequently employ reinforcement learning (sec. 2.3.2, p. 29) for their dynamic portfolio approach (see sec. 2.5.3, p. 42). They achieve 43% speed-up with respect to the fastest single algorithm in their portfolio [188]. This is QUICKSORT, which can itself be adapted by configuring its strategy for selecting a *pivot element*, and which is not even trivial to analyze in isolation [227]. Algorithm selection can also be performed for other fundamental algorithms, e.g., matrix multiplication [173, 207].[26]

Problem Solving Environments

Problem solving environments (PSEs) are all software systems that solve a specific class of computational problems. More concretely, they offer several solution alternatives and provide a *"language natural for the problem class"* [148, p. 2] to let users abstract from any concrete solution algorithm. This is still a very broad definition, e.g., it subsumes simulation software and scientific computing environments like MATLAB [223].

Since PSEs usually provide more than one method for solving a problem, they have to *select* among these when the user has entered a problem, and hence they have to solve the ASP. This issue is addressed by systems like PYTHIA [324] or its successor, PYTHIA II [146]. PYTHIA II selects a single algorithm per problem instance, based on prior knowledge that is induced by machine learning. It provides a layered architecture (see sec. 4.4.1) and a database for the management of experimental setups, problem features, and performance data. It operates 'on top' of software components that are regarded as black boxes, i.e., they do not have to convey any internal information to the system. This makes it necessary to provide additional integration code, e.g., for executing the software with

[26] In addition, matrix multiplication is a popular test case for empirical tuning (p. 54), e.g., used in [25, 319, 329].

a given problem or for obtaining the desired performance measurements. Similarly, the introduction of new problem domains requires to add additional, customized database structures. The system is able to weigh multiple performance aspects, e.g., execution speed and accuracy of partial differential equation solvers [146]. The selection mapping is generated in two steps. First, a statistical method is used to *rank* algorithm performance for all problems and performance metrics. A human *knowledge engineer* then selects which results to use for inferring *rules* from the given ranks and problem features. PYTHIA II does so with an *inductive logic programming* technique: algorithm ranks and problem features are represented by logical expressions, from which rules like

$$hasFeatureA(x) \wedge hasFeatureB(x) \Rightarrow bestMethod(a)$$

with problem $x \in \mathbb{P}$ and algorithm $a \in \mathbb{A}$ can be inferred. End users query PYTHIA II's *recommender system* to manually select a suitable software package for a given task. The actual selection in PYTHIA II considers the given problem features and orders all matching rules by their relevance and the amount of empirical data that support them. After applying the rules and thereby selecting an algorithm, the algorithm parameters can be selected by linear regression. The overall approach requires a *"dense population of similar types of problems"* [146, p. 228–229]. MyPYTHIA extends PYTHIA II by providing web-based access to its features [147].

While PYTHIA II is flexible enough to allow the addition of new machine learning methods, software packages, or problem domains, it is mainly focused on guiding users in their *manual* selection of *numerical* software. Incremental learning is not supported and many important aspects of simulation algorithms, e.g., the impact of stochastic effects (see sec. 2.3.1), are not addressed explicitly.

PSEs can be implemented by component-based systems, e.g., as discussed in [99], where the large design space of linear algebra solvers is discussed and *combinations* of algorithms are considered. Algorithm selection in PSEs is usually invoked at the beginning of processing time, chooses *once* from a set of *monolithic-parametric* algorithms (or their combinations), and takes into account the features of the current problem *instance*.

Empirical Tuning

Empirical tuning is a method for hardware-dependent compiler optimization, in which the most suitable way of executing statements from a high-level programming language shall be selected. This is usually done by some kind of combinatorial search through the (potentially large) space of possible code optimizations, e.g., via 'unrolling' loops so that CPU registers are used more efficiently [342]. Many empirical tuning methods can be regarded as algorithm selection from a set of *generated* algorithms, which is done *once* at *compile time*.

Empirical tuning is realized by *automatic tuning systems* such as PHiPAC [317, 318, 319], which generate possible realizations of a task given in a high-level programming language, and then search for the best realization by evaluating their performance. They predominantly aim at avoiding time-consuming manual adaptations to existing scientific programs, which strive for peak performance on specific hardware architectures. Vuduc et al. distinguish empirical tuning systems by the time at which the ASP is solved (compile time, processing time, or both), by the solution method (e.g., searching), and by the unit of optimization [319]: *kernel-centric* methods tune the given code as such, whereas *compiler-centric* systems tune the compiler so that it applies the right combination of optimizations.

The dependency on hardware is particularly eminent in this domain of algorithm selection, so that systems like PHiPAC are tested across various platforms (e.g., Sun Ultra, MIPS, Pentium-II, PowerPC, Cray) [318]. PHiPAC was shown to deliver performance comparable to vendor-specific (i.e., manually optimized) implementations for matrix multiplication, and could even outperform those on several problem instances. Although Vuduc et al. conceded that *"[...] performance can be surprisingly difficult to model on modern cache-based superscalar architectures"*[319, p. 65], they successfully applied machine learning methods for algorithm selection at processing time [318].

Empirical tuning can be regarded as a *complementary* technique to selecting simulation algorithms:[27] overall performance is improved by transforming *all* algorithms to machine code that runs optimally on the given architecture. However, this does *not* alleviate the task of high-level algorithm selection. Vuduc et al. conclude that they *"[...] could not reasonably expect a general purpose compiler to know about all of the possible mathematical transformations or alternative algorithms and data structures for a given kernel; it is precisely these kinds of transformations that have yielded the highest performance for other important computational kernels [...]"* [319, p. 83–84].

In [338], empirical tuning is applied to parallelized variable reduction. An example for automatically tuned matrix multiplication is the linear algebra software ATLAS [329], which is optimized with respect to caching, instruction ordering, and loop overhead. It even exploits instruction-level parallelism by hiding the latency of slow operations. Similarly, collective inter-processor communication via the *message passing interface (MPI)* can be tuned automatically [311]. Here, execution time is affected by network latencies and platform specifics, the message size, and the number of processors involved. Apart from the algorithm as such, the segmentation size (i.e., the number of segments a larger message is split into) as well as buffer sizes play an important role. An automatically tuned variant could outperform a native MPI implementation by up to one order of magnitude. An adaptive variant could improve the performance by another 25% [311]. Brewer applies compile-time algorithm selection to sorting and stencil computation (a method to solve partial differential equations) [25]. Frigo and Johnson use an empirical-tuning approach that relies on dynamic programming to speed up Fast Fourier Transformation [87]. Their approach outperforms many specifically tuned libraries, and they point out that *"computer architectures have become so complex that manually optimizing software is difficult to the point of impracticality"* [87, p. 1384].

Recent work by Ansel et al. [7] shows how algorithm selection can be explicitly addressed by a programming language: their PetaBricks language allows to define multiple algorithms (named *rules*) for computing a single function (named *transform*). Rules may be restricted to work only for some inputs and may include recursive calls. The PetaBricks compiler determines a suitable strategy of executing the rules of a transform in parallel (if possible) and selects the rules and their parameters with the help of an automatic tuning system.

Meta-Learning

The field of *meta-learning* constitutes an interesting application domain of algorithm selection. It aims at learning which machine learning algorithm to select for which kind of problem. Since the application domain is learning, researchers predominantly apply learning methods to solve the ASP, hence the term *meta*-learning. As a specific form of the algorithm selection problem, meta-learning is particularly relevant for guiding non-expert users of machine learning and data mining applications. Consequently, it is considered in standard use cases [167, p. 11] and also motivates flexible software frameworks in this area [216, p. 5]. The problem is also known as the *selective superiority problem* [174, p. 275]. An overview on meta-learning from an algorithm selection viewpoint is given in [292], which also discusses applications to sorting (fundamental algorithms, p. 53), constraint satisfaction (computationally hard problems, p. 56), and other domains.

Research on meta-learning is often dissociated from work in other algorithm selection domains, so it comes at no surprise that rather similar concepts have been developed under different names. For example, algorithm portfolio techniques are strongly related to *ensemble learning*, i.e., methods that apply *multiple* learning algorithms in order to improve prediction accuracy [333, p. 250 et sqq.]. Popular ensemble learning schemes are *bagging* and *boosting*, where several instances of a simple learning scheme are used to jointly solve the actual learning problem. Thereby, each instance is focused on a specific part of the problem, e.g., those cases that were mis-classified most often so far. All approximation forms generated by the instances are then combined to a single form, usually by some kind of voting scheme. While bagging considers all learners to be equal, boosting allows to weigh the votes of the learners, e.g., corresponding to their overall success on the training set. From

[27] In fact, empirical tuning techniques are often tested with code for scientific simulations, e.g., in [342].

an algorithm selection perspective, bagging and boosting can be regarded as applying portfolios that contain multiple instances of an algorithm [202, 205]. A related method, *stacking*, employs an explicit meta-learner to learn which instance within the ensemble, i.e., the portfolio of learners, performs best under which circumstances. The dynamic algorithm portfolio selection implemented by the extended AOTA framework (see fig. 2.10, p. 44) could be regarded as such an approach [96], since the AOTA framework incrementally learns to trust one time allocator more than others.

Apart from such parallel developments, there are also some rather unique viewpoints on the problem that stem from the meta-learning community. These deserve special attention, as they could also be useful in other algorithm selection domains. In [257], Pfahringer et al. introduce *landmarking*, a method to improve meta-learning: instead of using common problem features like the number of attributes, the performance metrics of simple and fast learners are used as problem features, i.e., to decide which of the complex learning schemes to prefer. Such an approach exploits the *duality* between an algorithm and the problem to be solved [174]: while classical approaches to algorithm selection try to characterize algorithms with respect to the problems on which they perform well, it might also be insightful to characterize problems by the algorithms that perform well on them. Ipek et al. use a similar line of thought for characterizing the hardness of an approximation problem, i.e., to show that the function they try to approximate is non-linear [155, p. 13]. In a simulation context, one would necessarily have to reduce the problem size in order to obtain any solution fast enough, otherwise the determination of problem features would take longer than any good algorithm selection could amortize (the metareasoning-partition problem, see sec. 2.3.3, p. 33). Landmarking could therefore be realized by some form of *pre-simulation*, i.e., a test-wise simulation over a comparably small timespan *before* setting up the actual simulation. Such techniques have already been used to improve PDES partitioning (see sec. 1.3.2, p. 6), where the communication frequencies among model entities are often unknown [35].

Finally, meta-learning is focused on an application domain where algorithms are often grounded on reasoning via information theory, hence there are also analytical approaches to select suitable learning algorithms (see discussion in [174, p. 276–278]).

Computationally Hard Problems

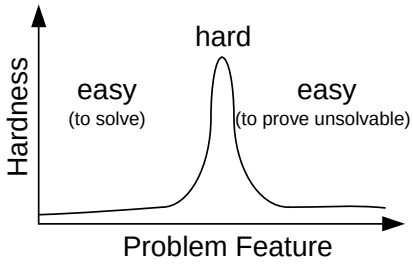

Figure 2.14.: Sketch of a phase transition.

There are exceptionally hard problems for which no efficient algorithm is known yet (e.g., [54, p. 439 et sqq.]). Nevertheless, some of these problems are quite relevant in practice, e.g., for planning [273] or graph partitioning [100]. The latter is required for model partitioning and hence is also relevant for distributed simulation (see fig. 1.4, p. 8).

Although such problems are known to be hard to solve *in general*, they have additional properties of practical importance. For example, the corresponding problem spaces are known to exhibit *phase transitions*, i.e., systematic shifts in problem hardness — only problems near a phase transition are hard to solve [37]. For hard decision problems, e.g., to decide whether a given logical formula is *satisfiable* by some variable assignment, a phase transition is also known as the *easy-hard-easy* pattern [273]. Rather unconstrained problems are often easy to solve, and it is equally easy to show that no solution can exist for very constrained problems. The problem instances that make the overall problem hard (in terms of worst-case performance, see sec. 2.2, p. 22) can be found in the middle of the phase transition (see fig. 2.14).

In [139], Hogg et al. relate algorithmic phase transitions to phase transitions from statistical physics, where a small change of a single parameter (e.g., temperature) can invoke the phase transition of the whole system into a new state with completely different properties, such as water that freezes to ice. They suggest that *"[...] the location of the phase transition point might provide a systematic*

basis for selecting the type of algorithm to use on a given problem" [139, p. 10]. Machine-learning is proposed to discover the parameters that determine the performance of the investigated search algorithms. Likewise, Guo envisages algorithm selection to distinguish between the hard and the easy problems [116, p. 19]. This would, for example, allow to select an exact algorithm whenever problem features indicate that this problem is far away from the phase transition, and to select an approximative algorithm otherwise.

Nevertheless, algorithm selection for hard problems has almost exclusively been tackled by algorithm portfolios, except for the approaches presented in [117, 145]. SATzilla [243, 336] is a very successful solver for the aforementioned satisfiability (SAT) problem. It is based on a portfolio of commonly used SAT-solvers. Satisfiability belongs to the general class of *constraint satisfaction problems (CSP)*, i.e., problems in which a variable assignment that satisfies some constraints has to be found. Other forms of CSPs are addressed in [114, 145, 149].

Simulation

The distinction between simulation environments and problem solving environments gets blurred when considering approaches that help with the selection of suitable numerical integrators. Numerical integrators are used for continuous simulation [34] and are usually selected so that they integrate a given set of differential equations as fast as possible, while keeping the error below a given threshold and maintaining numerical stability. At the same time, solving differential equations is considered as a task for scientific computing in general and is therefore addressed by PSEs like PYTHIA II [146] (see p. 53). Rice already discussed numerical integrator selection as a potential application domain when he introduced the algorithm selection problem [272, p. 77–82]. A more recent approach to select numerical integrators for simulation studies is briefly presented in [42] for the continuous simulation system Tornado [40]. It relies on reinforcement learning to identify desirable actions (i.e., algorithm selections) for certain model set-ups, characterized by the features of the simulation problem. However, neither specifics on the reinforcement learning technique as such are given, nor any performance results regarding its effectiveness.

Many adaptive synchronization protocols have been devised in the field of parallel and distributed discrete-event simulation (PDES) [52]. Some adaptation techniques are more sophisticated than simple heuristic rules that may or may not work for all kinds of models. For example, Wang and Tropper use reinforcement learning to find a suitable Time Window size, in order to optimally throttle down optimistic synchronization [322]. A similar approach is used to select and configure load balancing mechanisms during simulation execution [231].

Quaglia [267, 268] realized a simple form of continuous algorithm selection for speeding up single PDES runs: two replications are executed by different configurations in parallel, and the results of the fastest replicated *LP* are sent to *both* replications of the recipient. In other words, this approach employs a parallel portfolio approach in which the two algorithms of the portfolio interact with each other, i.e., they work together to solve the simulation problem. Ferscha et al. use a simple data mining strategy to analyze the sensitivity of different PDES synchronization protocols on problem features [79].

In [337], Xu and Tropper use machine learning to select between sequential and parallel discrete-event simulation. They employ 1-nearest neighbor search, i.e., they look up which algorithm to choose by considering the most similar scenario with respect to three problem features. Furthermore, they use the same technique to decide *how much* resources a parallel simulation should use to achieve an optimal execution time.

None the aforementioned approaches from the simulation community has been put into the general context of the algorithm selection problem. This also holds for the techniques to predict simulation algorithm performance, which are briefly surveyed in section 3.3 (p. 70).

Summary: Algorithm Selection Approaches

Even though the above sections span various fields of computing, algorithm selection is done in many other fields as well — be it implicitly or explicitly. Rice already proposed algorithm selection for operation system task schedulers [272]. It is also relevant for adaptive middleware [22, 230], planning [81], hardware design [262], and generally all application domains in which:

- a variety of algorithms exists
- the features of the problems to be solved vary strongly
- the interdependence between algorithm performance and problem features is non-trivial

Apart from rather ad-hoc and need-driven applications of algorithm selection, as described in the above subsections on fundamental algorithms and simulation, application domains often motivate certain families of approaches that are tailored to their specific requirements. Problem solving environments try to be adaptive via implementing single-algorithm selection for each problem instance. This requires a well-understood problem domain that motivates many features which indeed determine algorithm performance. Instances of hard problems, on the other hand, are also hard to analyze — here, portfolio approaches that exploit the properties of the algorithms' runtime distributions prevail. Since portfolio selection is also possible on purely statistical grounds, there is often no need to reason on the specifics of a concrete problem instance. Problem features are also neglected for many empirical tuning techniques. These shall select the most suitable machine code for the given hardware platform, so that they also often retreat to feature-less algorithm selection. The selection is focused on a single performance metric — execution speed — and is often done for a single problem instance only, i.e., these methods are oblivious. On the other hand, empirical tuning techniques usually *generate* the algorithms from which they select, which again increases the complexity of the task: they have to choose from a large amount of possible codes. Meta-learning relies on techniques similar to those applied to computationally hard problems, i.e., for selecting algorithm *sets*, but is able to include the existing domain knowledge to a much greater extent. In contrast to portfolios for hard problems, meta-learning allows the selected algorithms to work *together*. This often results in rather sophisticated approaches, e.g., the extended AOTA framework (sec. 2.5.3, p. 42). As figure 2.15 illustrates, both meta-learning and empirical tuning techniques can be used *in conjunction* with simulation algorithm selection: different selection mappings may be managed and chosen by a meta-learning scheme, while the use of an empirical tuning technique within the compiler would be completely transparent to the overall system[28]. Approaches from problem-solving environments and from the domain of hard computational problems are *both* important for simulation algorithm selection, depending on the available knowledge and data regarding a specific simulation problem domain.

2.8. Summary

This chapter introduced the algorithm selection problem (sec. 2.1) and described four basic approaches to its solution:

- *Analytical selection* by algorithmic complexity theory (sec. 2.2)
- *Empirical selection*,
  - by machine learning (sec. 2.3)
  - by adaptive software (sec. 2.4)
  - by algorithm portfolios (sec. 2.5)

[28] In fact, the *just-in-time compilation* techniques used in some Java runtime environments can be regarded as empirical tuning as well [319, p. 89].

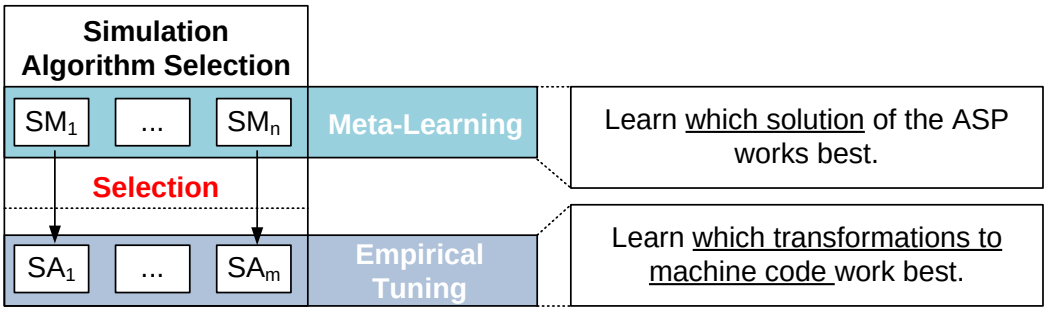

Figure 2.15.: Complementary methods for simulation algorithm selection. Meta-learners operate on selection mappings $SM_1, \ldots, SM_n$, i.e., they learn which of the ASP solutions performs best. This allows to identify the most promising ASP solution technique for a given simulation domain. Empirical tuning optimizes the execution of each single simulation algorithm $SA_1, \ldots, SA_m$, i.e., it learns how to make the most efficient use of the available hardware.

Machine learning and algorithm portfolio selection are based on the same fundamental techniques for data analysis, but differ in the *selection granularity* with which they solve an ASP: approaches from machine learning usually perform *single-selection*, whereas classic portfolio theory only allows to select a *set* of promising algorithms (sec. 2.6.1, p. 48). On the other hand, such portfolio approaches do not rely on any problem features. This eliminates the considerable effort required to identify the most suitable features for an ASP, which is indeed a challenging problem in itself (def. 2.1.3, p. 15). Recently, several approaches have been presented that blur the line between portfolio approaches and machine learning, e.g., [96, 205].

Adaptive software does not prescribe a selection or problem granularity, but focuses on approaches that solve the ASP *continuously*, and also apply solutions in that manner: autonomous software shall *optimize* itself automatically. Meta-heuristics (p. 20) are suggested for this task [193]. Some of them can be realized by some form of (decentralized) multi-agent system.[29] On the other hand, the specific algorithm selection technique is often neglected by adaptive software research, which is more focused on providing the *features* any adaptive software mechanism needs, e.g., component discovery or reflection (see sec. 2.4.3).

Only these empirical approaches are likely to be automatable. None of them offers a fundamental theoretical model of algorithm performance, from which a good selection mapping might be deducible — the mappings are derived from empirical data only, i.e., all three empirical approaches are prone to inductive fallacies (see sec. 1.4, p. 8). Consequently, the next chapter deals with the difficulties and pitfalls in collecting empirical data on simulation algorithm performance and discusses how this data can be analyzed, e.g., for predicting simulator performance (see sec. 2.3, p. 24).

Unsurprisingly, there is no silver bullet to algorithm selection — section 2.7 illustrates that there is not even a coherent research community dedicated to this issue. Following the approach of Sutton [298], who defined the field of reinforcement learning by the *problem* to be solved, instead of characterizing the techniques that solve it, *all* the above approaches belong to the field of algorithm selection because they solve some aspect of the ASP. The various dimensions discussed in section 2.6 should help to compare approaches from any of these communities, and to decide which are the most promising in a simulation context.

[29] For example, one could implement a genetic algorithm as a MAS by suitably defining the agents' environment and their interactions, similar to the Echo system described in [141].

3. Simulation Algorithm Performance Analysis

> *There are two ways of acquiring knowledge, one through reason, the other by experiment. Argument reaches a conclusion and compels us to admit it, but it neither makes us certain nor so annihilates doubt that the mind rests calm in the intuition of truth, unless it finds this certitude by way of experience.*
> Roger Bacon, *On Experimental Science*, 1268

Chapter 2 illustrates that the search for a good selection mapping depends on reliable performance data. To collect such data efficiently and to ensure that it is indeed trustworthy involves a whole range of additional techniques, which will be briefly discussed in the following: *"Unfortunately, as many researchers have already discovered, the field of experimental analysis is fraught with pitfalls"* [163, p. 216].

The chapter closes with a survey of performance analysis methods that are used in the domain of simulation, focused on performance prediction techniques for parallel and distributed simulation algorithms (some of which have already been detailed in [65]). As discussed in section 2.3, performance prediction is closely related to algorithm selection: an accurate prediction of simulator performance allows to select the most suitable algorithm beforehand, and hence solves the ASP.

3.1. Challenges in Experimental Algorithmics

Methodological guidance for the collection and analysis of algorithmic performance data comes from the field of *experimental algorithmics* [229]. The view that computer science is *empirical*, however, is much older than the notion of experimental algorithmics. For example, Newell and Simon maintain that many fundamental advances in artificial intelligence, a field subsuming several of the methods discussed in chapter 2, have been mainly driven by empirical results [237]. Moret [235] distinguishes experimental algorithmics from the experimental branches of other (natural) sciences, insofar as the former employ experiments because algorithm performance is often *too hard* to analyze in theory only, whereas latter resort to experimentation without alternative sources of knowledge. He motivates a focus on *implemented* algorithms and their *empirical* performance by giving various examples, e.g., for misleading results from computational complexity theory.

The importance of experiments has already been discussed in section 2.2: theoretical analysis is usually unable to fully characterize all relevant performance aspects, so that an empirical approach is mandatory. Apart from some imperatives concerned with scientific rigor in general, Johnson [163] advises algorithm experimenters to:

- Use reasonably efficient implementations.
- Ensure reproducibility.
- Ensure comparability.
- Use efficient and effective experimental designs.

Section 3.2 discusses the last advice in more detail. The other three advices seem quite straightforward, but are in fact hard to fulfill. However, the problems of lacking comparability and reproducibility

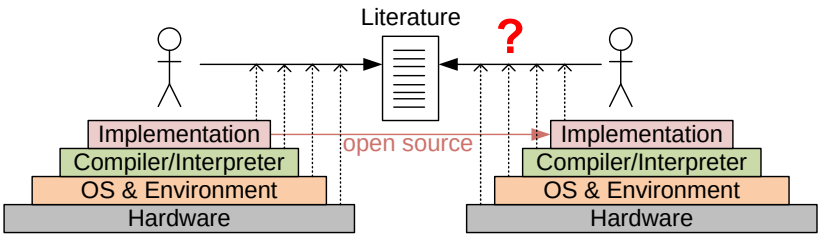

Figure 3.1.: Potential sources of re-implementation bias: all lower abstraction layers may affect the performance of an implementation (dotted arrows), both on the side of the initial developer and anyone who re-implements the algorithms. Publishing the source code allows others to exclude the implementation bias (red arrow), but not necessarily the bias due to operating system and hardware. In case of Java programs (and other interpreted languages), the Java runtime environment also plays a significant role (orange layer).

are not unique to the field of modeling and simulation; similar issues arise in other domains of computer science as well, e.g., in research on computer systems — where the authors of a survey were *"[...] dismayed at the lack of rigor and comparability of the published work"* [290, p. 14].

3.1.1. Efficient Implementations and Comparability

Using 'reasonably efficient' algorithm implementations seems self-evident, but it also means that all algorithms under consideration have to be realized with the same care and effort. For typical *horse race* publications in simulation, i.e., contributions that introduce a new algorithm and show where it outperforms existing methods [163, p. 216], this ideal is hard to achieve. Oftentimes, researchers have to re-implement existing methods based on descriptions from the literature, due to a lack of compatibility between programming languages, operating systems, and hardware platforms. As figure 3.1 illustrates, any performance measurements on a re-implementation may be biased and are unlikely to exhibit exactly the same characteristics as their original counterparts. Furthermore, algorithms are usually described verbally or with pseudo-code in the literature, instead of providing the lengthy source code. The more abstract and general an algorithm's description, the more error-prone is the process of re-implementing it. This problem might be alleviated by demanding that the algorithms in question have to be accessible alongside the scholarly publication, e.g., in the form of supplementary material. Hafer and Kirkpatrick put it bluntly: *"Only the original code provides enough detail for replication. Pseudocode of an algorithm is insufficient"* [120, p. 127].

There is also a subjective bias involved, since researchers may be inclined to spend much time profiling and fine-tuning their own algorithms, whereas competing algorithms are re-implemented in a plain and straightforward manner. Researchers may also tend to focus their performance analysis on those regions of the problem space where the newly developed algorithm performs best. This is only natural, considering that new algorithms are often designed to achieve better performance for a specific class of problems — but it may still distort the overall picture. All these factors may exaggerate the performance advantage that a new algorithm has over established methods, which in turn may lead to numerous (costly) re-implementations of the new method, although it is in fact unremarkable.

Some algorithms may exploit specific hardware features that are not available on other architectures, or can only be emulated at great performance cost. For example, GeorgiaTech TimeWarp (GTW) 2.0, a well-known simulation system based on the Time Warp synchronization protocol, exploits the features of shared-memory multiprocessors [53]. Porting it to another architecture requires to introduce new algorithms, i.e., it would eventually result in a *new* version of GTW, which is hard to compare to the former. More recent approaches face similar issues (e.g., [209]).

All these problems explain the focus on machine- or platform-dependent performance data, a viewpoint taken by almost all algorithm selection approaches (see app. A.3, p. 221). Nevertheless, it is sometimes possible to come up with hardware-independent performance measures, e.g., for search algorithms one can count the number of steps until a good solution is found [114, 145]. Similar metrics

exist for some simulation algorithms, e.g., one could observe the relative number of straggler events that occur during an optimistic simulation. Such rather abstract measures can be very useful, but the experimenter has to take care of two aspects:

1. Degree of hardware independence.
2. Correlation with real-world performance.

For example, the number of rollbacks might be independent of the hardware architecture as such, but it may vary when the relative speeds of network connection and CPUs are altered. If messages are delivered faster but processed at the same speed, this may reduce the number of roll-backs. Moreover, it has to be ensured that such indirect metrics correlate strongly with the performance metric that actually matters. For example, new synchronization protocols shall ultimately speed up the *overall* parallel and distributed simulation; if roll-back reduction is achieved by an algorithm that hurts overall performance, e.g., by blocking LP execution for too long or by being too expensive, just counting roll-backs will not uncover the algorithm's shortcomings.

All this speaks for the collection of performance data from the machine on which the algorithm selection shall ultimately be conducted. Good experimental setups may allow to transfer these findings to rather similar machines, but any other knowledge transfer based on empirical results will at least require some form of *calibration*, i.e., to conduct experiments on the target machine in order to correctly interpret and scale the performance data collected on other machines in the past.

3.1.2. Reproducibility

Reproducibility is necessary for falsification, and is hence an essential property of any scientific experiment (see sec. 1.1). The more complex the algorithms, the more error-prone is the process of evaluating and comparing them. Johnson proposes a *"broader notion of reproducibility"* [163, p. 229] for experimental algorithmics: a result is regarded reproducible if the performance evaluation of a re-implemented algorithm under *similar* circumstances (hardware, operating system, etc.) supports the former results and does not contradict them. This definition ensures that results are not rejected on the grounds of minor deviations — as figure 3.1 shows, it is hardly possible to achieve *exactly* the same results, and this even holds for experiments on the same machine (see below). To estimate and, if necessary, to document such external factors and their impact is the responsibility of the experimenter.

Hardware, Operating System, and Execution Environment

The exact reproduction of a single simulation run is practically impossible, as it depends on the state of the operating system and all involved hardware components like caches, which are also used by other programs running on the same computer. Software updates or hardware failures may further impact the machine's performance at any time. Eliminating all these factors is therefore considered infeasible [235]. The situation is even worse in parallel and distributed simulation, where hardware is either quite complex (e.g., in supercomputing) or includes additional devices for network communication, such as routers or switches.

A common way to characterize the hardware used in experiments is to provide the CPU model, the amount of available random-access memory (RAM), and additional *benchmark* scores, i.e., aggregated performance results for some widely used and standardized tasks that are designed to compare hardware platforms, operating systems, and so on. The experimenter has to carefully choose the benchmark that is most similar to the workload imposed by the algorithms under consideration. For example, there are several SPEC benchmarks for Java [293], some of which aim at rather general performance characterizations while others focus on features of particular importance in specific domains, e.g., business applications. For CPU-intensive operations, which are usually more relevant in scientific computing and simulation, Java SciMark [263] is a viable alternative. The description of operating system and execution environment is usually limited to product name and version number.

Problem Instances

Besides describing the infrastructure under which an experiment has been conducted, it is equally important to characterize the concrete problems to which the algorithms were applied. As already discussed in section 2.6.1 (p. 48), a major distinction is that between *realistic* and *synthetic* performance data, generated by experimenting with *realistic* and *synthetic* problem instances respectively. Realistic problem instances are advantageous because they are by definition representative of real-world problems and their practical relevance is therefore undisputed. On the other hand, algorithm evaluations for *new* problem domains can usually not rely on existing problems, e.g., if the aim is to explore a region of the problem space that is deemed important in the future. Such situations call for the generation of synthetic problems, which also has some additional advantages when it comes to the automated performance analysis of algorithms. For example, synthetic problems can be constructed to have some known solution, so that they can be used to detect implementation faults. A more detailed discussion on the merits and drawbacks of either kind of problem instance is given in section 7.3.1. In any case, the experimenter has to ensure that the problem instances are reproducible, either by specifying them explicitly or by describing the generation mechanism that creates them.

Stochasticity

Stochastic elements may be used on many levels, e.g., to create synthetic problem instances. They play an important role in discrete-event simulation, where they are used to calculate, for example, inter-arrival times or reactions occurrences (see sec. 1.3.1, p. 4). Reproducibility makes it necessary to detail the method with which these stochastic elements have been realized (e.g., [101]).

True randomness is hard to achieve on a deterministic machine like a computer. Although there are sources of 'true' random numbers occurring in nature (e.g., [119]), most problems are solved by relying on *(pseudo-) random number generators (RNGs)*, i.e., deterministic algorithms that generate streams of seemingly random numbers. RNGs are an important class of algorithms and have therefore received much attention over the past decades (e.g., [183, 192, 195]).

An RNG can be regarded as a very simple finite state machine, i.e., it has a finite number of states and an *iteration function* that defines the successor of each state. Typically, no input is considered: a new random number is generated by interpreting a part of the current state as a random number, usually normalized to the unit interval $[0, 1)$ by an *output function*. Such uniformly distributed numbers in $[0, 1)$ can then be transformed to various probability distributions, as required by the model or the simulator. Finally, the iteration function is applied to reach the successor state.

As an illustrative example, consider the popular family of *linear congruential generators (LCGs)* with an iteration function of the form:

$$x_{i+1} = a \cdot x_i + b \mod c \tag{3.1}$$

where x_i is the current state, x_{i+1} is the next state, and $\frac{x_i}{c}$ could be an output function.[1] The parameters a, b, and c are chosen to maximize the apparent randomness of the output, since correlated random numbers may invalidate simulation results (e.g., as shown in [115]). The initial state x_0 of an RNG is called its *seed*. Setting it determines the exact sequence of numbers that is generated, and is therefore needed to reproduce the simulation results.

An important figure of merit is the *period* of an RNG. It denotes the amount of iterations until the RNG reaches a prior state. Being a deterministic state machine, it repeats the generated sequence of numbers from then on. Each state transition *within* an RNG's period can be regarded as drawing a state randomly without replacement, which may introduce a bias even though the period itself is not exceeded. For example, the rule of thumb for LCGs with period n is to limit the generation to $\sqrt{n}$ random numbers [125, p. 492]. Still, periods of modern RNGs are very large, e.g., the well-known Mersenne Twister [225] has a period of $2^{19937} - 1$, so that this restriction is rarely a problem in

[1] Actual output functions are less naïve and only use specific parts of an LCG's state, in order to avoid non-random patterns in their output (i.e., correlations).

practice. More severe problems arise in parallel and distributed simulation, where random number generation is decentralized and *parallel* streams of random numbers need to be generated, for which several techniques exist (e.g., [222]). Testing the qualities of such parallel RNGs is often done by using them in distributed simulations of models for which an exact analytical behavior is known [294].

A related aspect is the seed size of an RNG, which defines the size of its state space and hence determines the maximum number of distinct number sequences that can be produced. The larger the seed size, the more distinct trajectories can occur in a stochastic simulation. For example, if x in equation 3.1 is represented by a 32-bit integer, this means that at most 2^{32} different number sequences can be used by a stochastic simulation, to simulate at most 2^{32} different trajectories. As discussed in [218], such limitations may bias the results of stochastic computations. A fundamental bias also stems from the mathematical form of the iteration function as such, e.g., Marsaglia could prove a relation between the modulo-parameter c of LCGs (see eq. 3.1) and the number of hyperplanes onto which most random tuples of LCG-generated numbers fall [217]. Such bias is hard to avoid in practice, as all RNGs have certain structural properties that lead to correlations of some kind. These correlations are often elusive and hard to derive analytically, so that RNG developers rely on empirical (i.e., statistical) tests [196, 278, 294]. Since there is no dominating (i.e., best) RNG and it is hard to tell how a given RNG affects results [125], the selection of RNG algorithms can be regarded as a domain where automation is *unlikely* to succeed. Situations in which RNG correlations affect the validity of simulation outcomes are very likely un-learnable (in the sense of [313], see sec. 2.3.3, p. 33); the recommended course of action is simply to reproduce results with distinct RNGs [125, p. 502]. Johnson puts it more boldly and advises us to *"never trust a random number generator"* [163, p. 247]. As illustrated in [226], even today's RNGs may produce strongly correlated output if they are initialized carelessly, so that the fast generation of high-quality pseudo-random numbers is still an area of research.

To ensure reproducibility in the broad sense, it is sufficient to define the algorithm that is used to generate the pseudo-random numbers. However, there are practical situations in which the exact course of a stochastic simulation run needs to be reproduced, e.g., when a more detailed observation of a concrete trajectory is desired. This requires to specify the exact implementation of the RNG as well as the seed with which it was initialized. Additional challenges arise in case of parallel and distributed simulation, where multiple RNG seeds are needed for initialization and have to be stored.

3.1.3. Simulation Experiment Descriptions

While section 3.1.1 illustrated that a trustworthy comparison of algorithms is quite challenging, section 3.1.2 named the most important aspects that have to be detailed in any experimental study on algorithm performance:

- hardware, operating system, language, and compiler
- algorithms (including random number generators)
- problems (and their generation method)

The large scale of the experiments required to gather sufficient data for empirical simulation algorithm selection makes it desirable to automate the description of these aspects. This can be done by creating experiment descriptions which are tailored to an existing simulation system (e.g., as it is currently done in JAMES II [68]), by general description languages (e.g., as proposed in [21]), or by application-specific approaches (e.g., for computational systems biology [185]).

However, to date there is no universally accepted standard way of describing simulation experiments. There are presumably several reasons for that. Firstly, the notion of an experiment can be regarded on different levels, e.g., one could regard a single simulation run as an experiment on a model, or instead regard an experiment as one or *more* simulation runs that aim at answering a given question. For example, single stochastic simulation runs rarely answer a question, as they require replications to

account for stochasticity. In the following, a simulation *experiment* (or just experiment) is considered to subsume one or more simulation runs.

Secondly, experiment descriptions are created for very different reasons. Approaches like SED-ML [185] aim at a very broad kind of reproducibility, i.e., to reconstruct similar simulation outcomes across different simulation environments. While this level of detail may be sufficient for exchanging general simulation experiment set-ups more easily, it is unsuitable for experimental algorithmics, since it does not explicitly specify all relevant performance factors, such as the hardware or the specific implementations that have been used.

3.2. Experiment Design

Apart from making sure that performance experiments can be reproduced by oneself and others and give trustworthy results, there are also various techniques to *design* experiments for efficiency. Efficient experiments consume as few resources as possible to answer the questions associated with them. The importance of good experiment designs in order to yield sufficient performance data, which can then be used to address the algorithm selection problem, has already been highlighted by others (e.g., in [311]). Since even statistically significant results are likely to be *wrong* when deduced from very small samples, biased set-ups, or a large set of hypotheses [154], sound experimental set-ups are a necessary condition for any kind of empirical study.

This section briefly surveys some design techniques that are applicable to experimental algorithmics. Policies for the multi-armed bandit problem have already been discussed in section 2.3.2; they can also be regarded as experiment design methods in that they steer sequential experiments toward the most promising options (e.g., simulation algorithms).

3.2.1. Variance Reduction

Results from stochastic simulations have to be regarded as samples from random variables, which are associated with probability distributions that are usually unknown. The goal of stochastic simulation experiments is often to investigate the properties of such distributions, e.g., to estimate the mean of a certain model variable or its standard deviation. Similarly, many algorithm performance measures — e.g., execution speed — are influenced by hardly controllable factors and are therefore often considered as samples from a random variable as well (see discussion in sec. 2.3.1, p. 25, and the empirical runtime distributions shown in fig. 2.11, p. 46). In the same sense that simulation runs need to be replicated for estimating the probability distribution of a random model variable, they need to be replicated for estimating the probability distribution of a performance measurement that is considered random to some extent, e.g., the execution time of a simulation algorithm. The process of estimating a performance distribution with sufficient accuracy may require numerous samples, each of them requiring an additional simulation run. This is particularly true if the random variable under investigation exhibits a high variance, i.e., its values deviate strongly from each other. Techniques for *variance reduction* aim at reducing the variance of an observed variable, so that the estimation of distribution properties, e.g., the mean, takes less samples, and therefore less simulation runs and less CPU time. Variance reduction is well-established [192, p. 577 et sqq.] and increases the *statistical efficiency* of a simulation experiment, i.e., to consume less resources while still attaining the desired statistical significance (e.g., in terms of confidence intervals). Two concrete examples of viable variance reduction techniques are given in the following, the first one being easily automated, while the second one requires further (manual) investigation.

Common Random Numbers

The most straightforward variance reduction method for experimental algorithmics is to let two independent simulation runs rely on *common random numbers (CRN)*. One merely needs to initialize the simulation runs with the same RNG and equal seeds. When the same stochastic simulation algorithm

is used — or even another one that makes equal use of random numbers — this lets both executions compute the same trajectory, i.e., it ensures that the computational load imposed by the simulation problem is the same for both. In case of non-stochastic simulation algorithms that execute a stochastic model, this method can be safely applied to compare any two algorithms with each other, thereby only (re-)initializing the model with equal RNGs and seeds. McGeoch suggests to use CRN *"[...] whenever algorithms are being compared, and there is reason to believe that their performance might be positively correlated with respect to input instances"* [227, p. 201–202].

While CRN is a straightforward concept and often facilitates a fair comparison of algorithms, it may be challenging to implement and only works if the simulators make *equal* use of random numbers, otherwise it may even increase the variance (see discussion in [192, p. 580 et sqq.]). Furthermore, it should be clear that comparing the performance of two simulation algorithms on a single trajectory is usually not sufficient: if the model trajectories themselves have some impact on simulator performance, a comparison based on a single trajectory will not capture the full picture. The relative performance of both algorithms might be quite different when they are evaluated on other trajectories. This pitfall can be avoided by comparing simulation algorithm performance on a *set* of RNG seeds.

Control Variates

Another interesting technique is the use of a so-called control variate. For example, let R_a^x be the random variable that denotes the run time measured for algorithm $a \in \mathbb{A}$ on a simulation problem $x \in \mathbb{P}$. The variance when sampling R_a^x, i.e., repeatedly executing simulation algorithm a on a problem instance of x and measuring its execution time, can be reduced with the help of another random variable S_a^x, which is known to be correlated with R_a^x and of which the expected value $E[S_a^x] = s_a^x$ is also known. The basic idea is to adjust, for each simulation run, the sampled value from R_a^x with respect to the sampled value from S_a^x. As S_a^x now exerts some form of control on R_a^x, it is said to be the *control variate*. Formally, one constructs a new random variable T_a^x that has a lower variance than R_a^x and takes into account S_a^x:

$$T_a^x = R_a^x - \alpha(S_a^x - s_a^x) \tag{3.2}$$

where α is the factor that determines how much control is given to S_a^x for adjusting the sampled values of R_a^x. The sign of α is positive if R_a^x and S_a^x are positively correlated, and negative otherwise. Hence, in case of positive correlation, equation 3.2 *decreases* the value of T_a^x if the sampled value from S_a^x exceeds the expected value s_a^x. If the sampled value from S_a^x is smaller than the expected value, equation 3.2 *increases* the value of T_a^x instead. Intuitively, the scaling factor α should be larger the more R_a^x is correlated with S_a^x — the optimal value for α can be derived [192, p. 601] to be:

$$\frac{Cov[R_a^x, S_a^x]}{Var[S_a^x]} \tag{3.3}$$

where Cov and Var denote co-variance and variance, respectively. However, these values are usually *unknown*, so that they have to be replaced by estimates in practice (see [192, p. 602–603] for a more detailed discussion). Furthermore, the method requires prior knowledge on the control variate S_a^x, i.e., its expected value s_a^x (or at least a very close estimate) — which makes it hard to automate. For algorithm performance analysis, McGeoch suggests to consider the performance of simpler algorithms as potential control variates [227, p. 202]. This would relate the performance of simple algorithms on the same problem instance to the performance of more sophisticated ones, an idea that is rather similar to landmarking in meta-learning (see sec. 2.7, p. 55).

Further Techniques

There are several other variance reduction techniques that are similar to the control variate approach in that they allow to construct new unbiased estimators by integrating additional information. For example, one may consider the *conditional expectation* of the original estimator, i.e., restrict the

sampling to certain situations [227]. Alternatively, one may try to construct pairs of problem instances on which the sampled values are *negatively* correlated, i.e., make use of *antithetic variates*. All these techniques help to reduce the number of samples required to arrive at good estimates for the expected value of the random variable in question. In case of experimental algorithmics, they allow a much faster performance analysis: less simulation runs are required to arrive at a good estimate for a performance measurement, e.g., the average run time of an algorithm. A survey on the topic can be found in [192, p. 577 et sqq.], the application to experimental algorithmics is detailed in [227]. Finally, it should be noted that variance reduction is not always beneficial. Some meta-modeling approaches (see sec. 3.2.2), e.g., stochastic kriging [6], consider the observed variance in their calculations, and are hence mistaken when the variance is reduced artificially. Furthermore, performance variance might be an interesting aspect in itself, e.g., when constructing algorithm portfolios (see sec. 2.5.2, p. 40).

3.2.2. Optimization, Sensitivity Analysis, and Meta-Modeling

While variance reduction helps to minimize the amount of samples that are required for a good estimate of *one* algorithm's performance on a *single* simulation problem, other techniques are required that select the simulation problems $x_i \in \mathbb{P}$ to be considered. The most simple technique, *parameter scanning*, is to let the user specify manually which problems are of interest.

The suitability of a technique depends on the question to be answered by the performance experiment. For example, algorithms for black-box optimization (see sec. 2.1.3, p. 20) could be used to find a parameter combination for an algorithm under which it performs best, with respect to a set of simulation problems $P \subset \mathbb{P}$.

Techniques for *sensitivity analysis* can be used to select simulation problems $\{x_1, \ldots, x_n\} \subset \mathbb{P}$ for a good quantification of the performance impact that some problem features $\{f_{x_1}, \ldots, f_{x_n}\} \subset \mathbb{F} = \mathbb{R}^m$ impose on an algorithm $a \in \mathbb{A}$ (cf. fig. 2.1, p. 14). Thereby, special attention is paid to *interacting* problem features. For example, the size of a simulation model and a certain structural property, e.g., the connectedness of the model entities, may strongly affect an algorithm's performance if *both* are increased simultaneously. Such interactions may occur between arbitrary numbers of feature space dimensions, and they may differ from algorithm to algorithm. Knowing all feature interactions and their performance impact for all algorithms would essentially solve the best features for algorithms problem (cf. def. 2.1.3, p. 15), as it allows to identify those features that are most relevant for the given set of algorithms $\mathbb{A}_0 \subseteq \mathbb{A}$. However, sensitivity analysis is a challenging discipline that requires some computational efforts. Consider a performance analysis experiment to identify the interactions between three simulation problem features, i.e., $(f_x^1, f_x^2, f_x^3) \in \mathbb{F} = \mathbb{R}^3$ for any simulation problem $x \in \mathbb{P}$. To find out about the interactions among all three feature dimensions, we have to consider at least two values for each of them, so that the set $\mathcal{F}$ of all feature combinations can be defined as

$$\mathcal{F} = \{f_{low}^1, f_{high}^1\} \times \{f_{low}^2, f_{high}^2\} \times \{f_{low}^3, f_{high}^3\} \tag{3.4}$$

$\mathcal{F}$ contains $2^3 = 8$ elements and grows *exponentially* with the number of features. Each feature combination stands for at least one simulation problem from $\mathbb{P}$ that exhibits these features.[2] Hence, the number of simulation problems to be considered grows exponentially as well. Each of the simulation problems has to be solved sufficiently often by *each* of the algorithms under comparison, so that reliable estimates on their average performance can be made (see sec. 3.2.1). Furthermore, equation 3.4 is still a rather naïve approach, as it implicitly presumes that feature interactions will be the same over the whole feature space, i.e., that it is sufficient to consider just two values for each dimension. While this problem can be alleviated by choosing representative values, the exponential growth of the data required by the *full-factorial* setup (eq. 3.4) can only be diminished by more sophisticated constructs, such as Plackett-Burman designs [259] or fractional factorials [192, p. 636 et sqq.]. These methods allow a trade-off between the number and the kind of detected interactions on one side, and the number of simulation problems to be analyzed on the other.

[2] Such a relation between problem features and simulation problems needs to be constructed carefully, and is usually implemented by specific benchmark models (see sec. 7.3.1, p. 145).

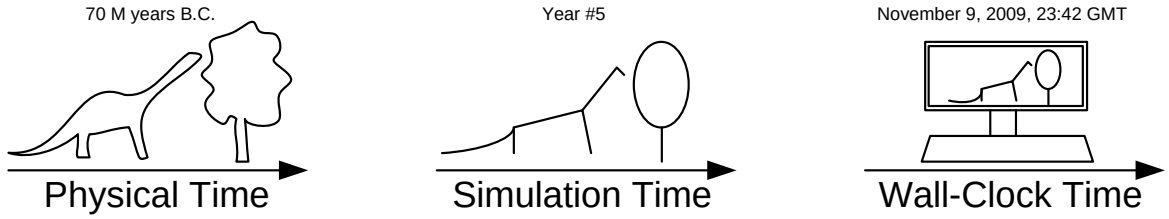

Figure 3.2.: Time scales in simulation: the *physical time* is the time of the original system (left), the *simulation time* is the time within the simulation model (middle), and the *wall-clock time* is the time at which the simulation is running (right); see [90, p. 27].

Finally, so-called *meta-modeling* techniques often require specific experiment designs for the efficient creation of *meta-models*. A meta-model relates the input parameters of a given simulation model to certain outcomes of its simulation. Meta-models are usually synthesized automatically and have a predetermined (but parameterizable) mathematical form, e.g., a polynomial function with variable coefficients. After conducting an experiment with the original simulation model, the coefficients of the meta-model are fitted to the simulation results of the original model. The meta-model can then be used to *predict* the outcomes of simulating the original model on formerly untested input parameters. Hence, meta-modeling can be regarded as constructing approximation models for simulation output (see sec. 2.3, p. 24). As a consequence, methods to design good experiments for meta-modeling, e.g., presented in [192, p. 643 et sqq.], may also be applicable to the algorithm selection problem: having accurate meta-models for algorithm performance allows to select the most suitable algorithm for a given problem.

So far, however, meta-modeling is predominantly used in the context of simulation-based optimization (e.g., [88]). Here, meta-models are used as replacements for original simulation models, in order to speed up the overall optimization process. Current methods, such as *stochastic kriging* [6], also allow to steer an experiment towards the most 'interesting' regions of the simulation problem space $\mathbb{P}$. On the other hand, the application of meta-modeling may be hampered because the underlying assumptions are hard to assert. For example, a kriging interpolation for algorithm performance would presuppose a specific relation between the distance of simulation problems (i.e., feature difference) and the metric to be interpolated (e.g., [47, p. 105 et sqq.]).

Besides designing experiments for algorithm performance analysis, many of the aforementioned techniques can also be used to *test* the simulation algorithms in question, i.e., to check if their implementation is erroneous. For example, an algorithm could exhibit a strong dependency on problem features that are deemed irrelevant by the developer. This may hint at a software bug. An introduction to simulation experiment design is presented in [192, p. 619 et sqq.]; more advanced techniques can be found in [127].

3.2.3. Further Aspects of Performance Experiments

Apart from decreasing the number of required simulation replications (sec. 3.2.1) and selecting the most interesting simulation problems (sec. 3.2.2), there is a wide range of additional techniques to increase the efficiency of simulation experiments. However, not all of them can be easily applied to experiments that are concerned with the *performance* of the simulation algorithms instead of the simulation outcomes as such.

For example, so-called *stopping rules* are used to quit the simulation when a certain condition is true. The most common and simple stopping rule makes a simulation stop when a certain amount of *simulation time* has gone by, which is the time span considered within the model (see fig. 3.2). Other rules may be more complex, e.g., to simulate until the output of a variable has reached an equilibrium, or to stop after some amount of *wall-clock time* (see fig. 3.2) has passed. Stopping a simulation run after some wall-clock time is similar to the notion of *censoring* that has been employed

by the AOTA framework for parallel algorithm portfolios (see sec. 2.5.3, p. 42): all other algorithms are stopped when the first one finds a solution. Nevertheless, most stopping rules are defined upon simulation variables, and hence cannot be transferred easily from ordinary simulation experiments to performance analysis. Furthermore, stopping rules interact with other aspects of experiment design, so that the effect of a certain rule on overall efficiency may have to be determined by prior analysis. For example, it is non-trivial to find a good compromise between executing a few long simulation runs that are sped up by parallel and distributed simulation on the one hand, and executing many short simulation runs as independent replications on the other hand [124].

Another way to speed up simulation experiments is to re-engineer the model under scrutiny. Sophisticated data sampling can achieve speed-ups of more than 100 for hardware simulations [26]. This is done by identifying the most relevant operations for which the hardware model shall be simulated. The basic trade-off is between the simulation speed and the accuracy of the input data, i.e., the representativeness of the selected operations. For simulation algorithm performance analysis, one could adopt similar methods to construct problems that are executed as fast as possible. The creation of benchmark models is further discussed in section 7.3.1.

All in all, experiment design can be a powerful tool to analyze simulation algorithm performance. However, the creation of good designs with known statistical properties is still complicated, due to the many interacting aspects. It still *"[...] requires the involvement of both experienced applied statisticians and experienced computer performance analysts"*[123, p. 1202], just as in former decades. For example, if simulation replications are deemed independent but produce correlated output, e.g., due to correlated pseudo-random numbers, this leads to a systematic underestimation of variance [123, p. 1215]. While an intentional variance reduction is often useful (sec. 3.2.1), an *unintentional* reduction leads to biased results.

3.3. Simulator Performance Analysis and Prediction

While sections 3.1 and 3.2 discuss the basic requirements and techniques for setting up meaningful and viable *experiments* for algorithm performance analysis, this section briefly surveys some of the existing performance analysis approaches, with a focus on methods that have already been applied to simulation. Performance prediction and analysis has been particularly popular in the domain of parallel and distributed discrete-event simulation (PDES): the performance improvement that often motivates the application of PDES is quite sensitive to problem features, hardware, and the (manually) selected algorithms that are used (see sec. 1.3.2, p. 6).

Similar to the PDES-centric overview of analysis methods presented in [65, p. 33–41] and to the categorization of algorithm selection methods in section 2.6.1 (p. 48), the methods applicable to simulator performance analysis are divided into two groups, depending on the extent to which they rely on empirical data:

- **Analytical**: Methods that do not take into account any empirical data when constructing the performance model. They may rely on empirical data only insofar as they feed it into their separate analytical model of the system, e.g., in form of parameter values.

- **Empirical**: Methods that fundamentally rely on empirical data, i.e., they use it to construct a performance model for a simulator (see discussion on meta-modeling, sec. 3.2.2, p. 68).

It should be noted that the general idea of predicting the performance of algorithms is anything but new, e.g., McGeoch refers to work from the 1980s that aims at predicting the solution quality of bin packing algorithms [227, p. 207]. Similarly, the importance of performance prediction in the context of scientifically approaching algorithm development has already been acknowledged more than a decade ago [142].

3.3.1. Analytical Methods

In [238], Nicol constructs a mathematical model of a parallel and distributed discrete-event simulation in order to analyze its *scalability*. In this context, scalability refers to the efficiency of the simulation when either some aspect of the problem (e.g., size) or the amount of used hardware is increased.

Efficiency, in turn, is characterized by *processor utilization*, i.e., the share of operations that would have also been necessary to carry out in case of a sequential simulation. After introducing several parameters, e.g., costs for event queue operations and communication rates among processors, the analysis yields a complex expression for a lower bound on processor utilization.

Scalability can now be defined more concretely by demanding that the lower bound on processor utilization has to remain constant (or to rise) if model size and number of processors are increased simultaneously, as defined by a growth function. This formalized notion of scalability is then used to prove that a certain synchronization protocol is scalable if all processors are connected by a hypercube[3], which nicely illustrates the strength of analytical performance prediction: if applied successfully, it allows very broad insights into the nature of an algorithm *as such*, i.e., analysis results are general and implementation-*independent*.

On the other hand, this analysis relies on many strong assumptions, e.g., a homogeneous set of processors and non-zero delays between events. It also focuses on *average* costs, making it hard to estimate the impact of dynamic aspects. Furthermore, Nicol concedes that such proofs might not be attainable for all kinds of algorithms: *"[...] synchonization behavior is frequently complicated, which makes it very difficult to analytically prove anything about performance executing large models on large machines"* [238, p. 4]. From this perspective, even the improvement of existing data structures, such as event queues, can be regarded *"[...] as sort of good news and bad news deal [...]"* [238, p. 11], since it may entail a revision of all proofs and derivations due to altered basic assumptions. All in all, analytical performance prediction basically has the same advantages and disadvantages as analytical approaches to algorithm selection (see discussion in sec. 2.2, p. 22).

In [118], Gupta et al. model processors interacting via the Time Warp synchronization protocol (see sec. 1.3.2, p. 6) as a continuous-time Markov chain (see sec. 2.3.2, p. 29). Formally deriving the transition rates between the states allows them to predict the runtime of simulating a synthetic benchmark model fairly accurately, i.e., within 10 percent of the actual execution time. However, some strong assumptions again hamper the broad applicability of the results: processors are homogeneous, time stamps are exponentially distributed, and communication between processors does not cost any time.

Cortellessa and Quaglia derive an upper bound on the execution time of an optimistic parallel and distributed simulation [45]. They assume that each local event queue holds a single event, that the processing of each event results in the generation of exactly one new event, and that idling times and message delay can be neglected. Similar to the approach presented in [118], they also prescribe the shape of the time increment distributions. They point out that the *"[...] efficiency of an optimistically synchronized parallel simulation is, in general, highly unpredictable, depending on features of the simulation model and of the hardware/software architecture"* [45, p. 1].

Other approaches make fewer general assumptions, but rely strongly on the specifics of the synchronization protocol that is used, e.g., [240]. A prediction method that is mathematically straightforward but instead relies heavily on realistic cost measurements for the main operations (e.g., event management, thread switching) is presented in [208]. Although the authors carefully measure the cost of each operation, the resulting prediction error still varies between 5 and 30 percent.

Similarly, the prediction approach of Juhasz et al. [168, 169] relies on extensive measurements of a prior sequential simulation run. In order to predict the performance gain of a parallel and distributed simulation approach on the *same* trajectory, the authors record the *event precedence graph* of the sequential execution, which is depicted in figure 3.3. For example, this allows to identify the *critical path*, which is the sequence of causally related events that takes the longest time to be computed. Intuitively, it is impossible to speed up any sequential execution *beyond* the time it takes to sequentially

[3]The assumption of having a hypercube is required for bounding the communication delay.

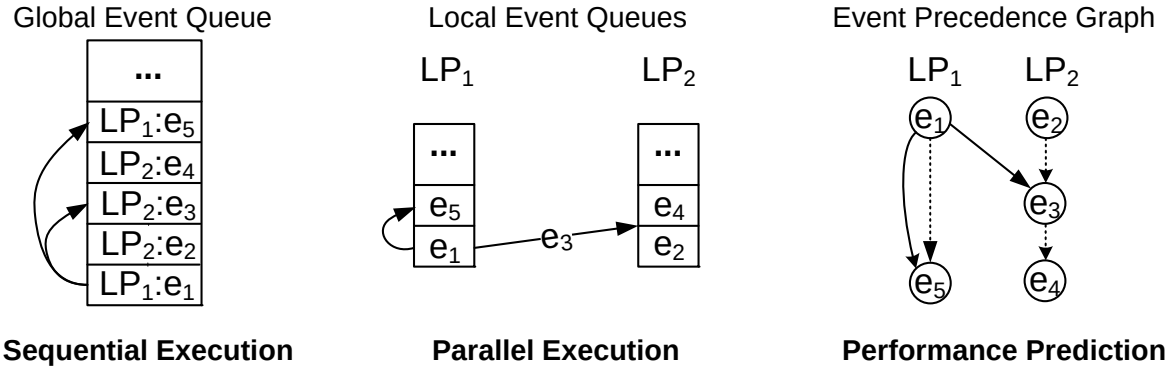

Figure 3.3.: Performance prediction via event precedence graphs [168]. The event precedence graph (on the right) is directed and acyclic. It combines event scheduling information (solid arrows: e_1 schedules e_3 and e_5) and causality constraints per logical process (dashed arrows: e.g., LP_2 has to execute e_3 before e_4). This information can be extracted from a sequential simulation (left) to predict the runtime of a parallel and distributed simulation (middle).

compute the critical path, as its computation cannot be parallelized any further — it is a natural upper bound on speed-up due to parallel execution.[4] The overall execution time of a parallel simulation can now be predicted by considering the event precedence graph and assuming that some events of the graph are computed on different processors. An additional mathematical model for communication cost complements the approach, with which Juhasz et al. now can predict the performance for different synchronization protocols, different communication costs, and different numbers of involved processors. In [302], the approach is extended to work for optimistic synchronization protocols as well. This strong reliance on the (empirically observed) event precedence graph, which is analyzed to come up with a prediction, makes this approach just as empirical as it is analytical — it could even be automated. However, such kinds of performance prediction require a prior sequential execution and an exhaustive analysis of the recorded results, so that the scalability to real-world problems might be an issue.

Instead of predicting the runtime of a parallel and distributed simulation method, and thus being able to identify the situations in which it is preferable over a sequential approach, other analyses focus on more abstract metrics of performance. For example, Heidelberger examines how the statistical efficiency of stochastic simulation experiments is influenced by various factors, including the number of processors, the available time, and the runtime performance of the available algorithms for parallel and distributed simulation [124]. Similarly, Nicol proposes to consider the *utility function* of a simulation user, i.e., how much the utility of the simulation increases with the size of the model to be simulated [239]. This links analytical performance analysis of parallel and distributed simulation with economic decision theory: is it more useful (i.e., efficient, inexpensive, etc.) to sequentially simulate a small model on each processor, or is it better to simulate one large model that is distributed over all processors? Nicol's analysis shows how this decision depends on the utility function of the user. Such utility functions are also used in financial portfolio theory (see sec. 2.5, p. 38), where the form of an investor's utility function has to be considered for a mathematical analysis. Similar approaches could be used to solve the ASP, considering a user's overall utility for each eligible simulation algorithm.

[4] As pointed out in [168, p. 11], it is theoretically possible that optimistic synchronization outperforms the critical path, due to speculative execution. Given that event A causes event B, B can usually only be computed *after* the execution of A is finished — following the critical path. However, if B is *likely* to be caused by A, it might already be executed before the execution of A is finished, so that the critical path is outperformed.

3.3.2. Empirical Methods

In contrast to analytical performance prediction methods, which mainly focus on parallel and distributed discrete-event simulation, empirical methods to predict simulator performance are often applicable to a much broader range of problems. If one is not constrained by the assumptions that are required for an analytical prediction, it seems natural to regard simulation software as just some arbitrary program for which performance shall be predicted.

For example, Sherwood et al. predict the execution speed of a program by identifying its so-called *basic blocks*, which are small sequences of instructions with single entry and exit points [288]. The frequency with which each block is executed during a time window is recorded in a so-called *basic block vector*. Different operation modes of the program are identified by clustering the recorded vectors to groups, which now represent similar program loads. Finally, the overall performance is predicted by simulating the execution of a typical representative from each cluster and aggregating the results. This technique is rather elaborate and time-consuming, but was shown to predict the number of required instructions with an average error of only 3 percent. Its execution time can be improved by meta-modeling (e.g., [155]).

The Prophesy system approaches performance prediction in a similar manner, although it is specifically focused on parallel programs [301]. Each program is decomposed into a hierarchy of building blocks: modules, functions and block units, which are *"of finer granularity than a function but coarser granularity than a basic block"* [301, p. 14], e.g., some nested loops. A sub-system of Prophesy parses and instruments the source code of the application, so that detailed performance data can be recorded. Performance data and structural information are then submitted to a database. Prophesy supports both automatic and manual performance modeling for each block, and it allows to analyze performance interaction between adjacent code blocks, i.e., code blocks that are executed one after another. For example, some code blocks may rely on the same data being in the cache and are hence executed faster in sequence than in isolation. Others may hamper each other's performance due to similar effects. Prophesy allows to aggregate the performance models of all these code blocks and to take these interactions into account. This reduces the average prediction error, e.g., from 35 to 14 percent in one test case.

A similar tool to ease performance analysis and prediction for parallel programs is PerfExplorer [150], which provides several data mining methods[5] to analyze and predict runtime performance. Similar to Prophesy, PerfExplorer allows a fine-grained collection of performance data by 'events', which may refer to the execution of anything from a single line of code to a whole library. The performance data is stored in a database [151] and can be analyzed by other tools as well, e.g., the Weka [327] data mining toolkit.

The Zenturio system [265] allows to set up various kinds of algorithm experiments for cluster and grid environments, and even features a custom experiment definition language, Zen. Zen directives can be distributed over several text files of arbitrary format. Zenturio analyzes them, changes them according to the directives, and then executes each set-up. The tool also supports storage and plotting of performance data. However, it is not simulation-specific, but instead dedicated to the functioning in specific environments for parallel computing (Globus [85] and PBS [126]). Furthermore, its experiment definitions are intertwined with various files, e.g., for building the code or scheduling the jobs. This makes it hard to archive experiment definitions, and also makes it difficult to define *distinct* experiments on the *same* code base. All in all, it is complicated to enforce the specification of the experiment aspects outlined in section 3.1.3 (p. 65).

The POEMS system [3] goes one step further and combines analytical models with simulation models and raw empirical data to predict the performance of parallel programs.[6] It relies on a custom specification language to combine those components, which, for example, may be processed by

[5] For example, clustering methods similar to those used by Sherwood et al. in [288].

[6] Similar to the approach of Juhasz et al. [168, 169, 302], POEMS is a borderline case with respect to the analytical-vs.-empirical categorization. It is assigned to the section on empirical methods because is does *not* presuppose any mathematical structure that is analyzed afterwards, as the approach of Juhasz et al. does (by relying on event precedence graphs, see fig. 3.3).

hardware or network simulators. Its authors claim that the *"[...] key innovation in POEMS is a methodology that makes it possible to compose multidomain, multiparadigm, and multiresolution component models into a coherent system model"* [3, p. 1027]. This results in rather accurate performance prediction with an error between 4 and 10 percent (for the sample application). On the other hand, the detailed modeling of a complex software cannot be automated easily and may require considerable additional effort. Nevertheless, the importance of simulation for predicting program performance has motivated new simulation tools that allow performance analysis in even more complex settings, e.g., GridSim [29].

There are many other generally applicable methods to analyze the performance of programs, e.g., to estimate the constants of their asymptotic performance (see sec. 2.2, p. 22) by relying on sophisticated experimental setups [82], or to express statistically significant differences between algorithm performances with logical expressions [270].

To experimentally analyze the performance of simulation algorithms, Balakrishnan et al. introduce the *Workload Specification Language (WSL)* [12], which allows to easily specify synthetic benchmark models for experimenting with parallel and distributed simulation algorithms. It allows to translate simulator-independent benchmark model specifications into simulator-specific code, which enables a fair comparison of parallel and distributed simulation systems even if their interfaces are incompatible.

In [79], Ferscha et al. conduct a full factorial experiment on various PDES execution schemes to investigate their sensitivity to certain model properties. In [17], Bauer Jr. et al. investigate the scalability of Time Warp on very large numbers of processors (more than 100.000). Following the same line of thought as in [3, 288], namely to predict program performance by simulating it,[7] one may either model and simulate the simulation algorithms themselves by additional tools [71, 103], or virtualize their execution for easier analysis, as proposed in [254].

In the application domains of simulation — such as computational biology — performance studies are mostly done ad-hoc, e.g., when a new simulation algorithm is presented. Performance results for the stochastic simulation algorithms (SSAs) introduced in section 1.3.1 (p. 4) are discussed in chapter 9.

3.4. Summary

Section 2 introduced the formal problem to be solved and then proceeded to outline possible solution techniques and related work, thereby following a top-down approach from the abstract problem definition to concrete examples for solutions. This chapter completes the background part by illustrating the challenges of collecting the valid empirical performance data that is necessary for automated algorithm selection, thereby following a bottom-up approach from practical problems in experimental algorithmics (sec. 3.1) over experiment design techniques (sec. 3.2) up to advanced analytical methods (sec. 3.3.1) and empirical methods (sec. 3.3.2) for performance analysis and prediction.

The issues discussed in section 3.1 (p. 61) are still topical. In 2002, Pawlikowski et al. surveyed more than 2.000 scientific papers that applied stochastic simulation in the domain of communication networks [250]. Over 75% of them do not deal properly with output analysis (e.g., averaging over multiple replications, calculating confidence intervals, etc.) or ignore common pitfalls, e.g., regarding random number generation. The authors also point out that *"[...] the real problem is that the vast majority of simulation experiments reported in telecommunication network literature are not repeatable"* [250, p. 137].

None of the methods discussed in section 3.3 has found widespread acceptance in the simulation research community so far; most developments have been ephemeral. In case of analytical methods, this might be due to the strong underlying assumptions that just do not seem realistic enough to support accurate predictions. The empirical methods, on the other hand, are often implemented in separate tools that are so general and sophisticated that simulation researchers may be deterred

[7] This has already been suggested by Rice in his 1976 paper on algorithm selection [272].

from applying them to their own problems. For example, POEMS brings along its own component-based modeling and simulation approach [3]. Prophesy and PerfExplorer are designed to analyze program performance on a fine-grained level, which may be unnecessary for many simulation algorithm developers. Similarly, a language for defining platform-independent benchmark models (as proposed in [12]) is all nice and well, but apparently requires too much effort to implement for each single simulation system.

A promising way to overcome the reluctance of using sophisticated performance analysis is to automate the whole process as much as possible. It requires a tight integration of performance analysis tools and simulation system, but might make performance analysis much more attractive in the future. Hence, the next chapters also discuss how automatic mechanisms for performance analysis can be integrated into the simulation system JAMES II. Their ultimate goal is to support automatic algorithm selection by efficiently delivering trustworthy performance data. Nevertheless, such mechanisms could also be useful in many other situations.

Part II.

Methods and Implementation

4. A Framework for Simulation Algorithm Selection

What I cannot create, I do not understand.
Richard P. Feynman

The background part surveyed methods for automated algorithm selection (ch. 2) and detailed the major challenges and techniques of empirically analyzing the performance of simulation algorithms (ch. 3). This part tackles the central issue of the thesis, namely the automatic selection of simulation algorithms. It mainly treats the construction of a prototypical simulation algorithm selection framework for the modeling and simulation framework JAMES II. The term *host system* helps to distinguish between both software systems: the host system of an algorithm selection mechanism is the software that contains the algorithms to be selected. In this sense, JAMES II can be regarded as the host (simulation) system of the framework to be developed.

This first chapter of part two starts out with analyzing the general requirements that arise from different use cases (sec. 4.1). It briefly sketches the core principles of JAMES II (sec. 4.2) and identifies the technical requirements for a simulation algorithm selection mechanism that is integrated into JAMES II (sec. 4.3). Section 4.4 first discusses the structure of some software systems that serve similar purposes (sec. 4.4.1), and then outlines the major components of the simulation algorithm selection framework. The latter draws upon the aspects previously presented in this chapter: use cases, JAMES II specifics, technical requirements, and related systems.

4.1. Requirements Analysis: Use Cases

At first, let us consider the use cases in which simulation algorithm selection may play a role. Figure 4.1 summarizes them in a *UML*[1] use case diagram.

The ultimate goal of the selection mechanism is to deliver an automatic configuration feature for the end users of a simulation system, i.e., anyone who aims at analyzing a model via simulation experiments. Here, the selection mechanism could be regarded as a black box or an oracle, which gives expert advice on a subject matter that the user is unsure about. Hence, the user interface to the mechanism should be easy to understand, and the mechanism's configuration shall be as simple as possible.

The second major target audience of such a tool are developers who work on simulation. As the selection mechanism basically considers empirical performance data to select the most suitable option for a given simulation problem, algorithm developers could use it to validate their latest enhancements, i.e., to check if these really have a positive impact on the *overall* performance. Consider the enhancement of a simulation algorithm $a_1 \in \mathbb{A}$ that makes the algorithm much faster for an important class of simulation problems $P_1 \subset \mathbb{P}$, while degrading its performance on a rather obscure class of problems $P_2 \subset \mathbb{P}$. While this enhancement is clearly beneficial if a_1 is the only available option in the system, the presence of *another* algorithm $a_2 \in \mathbb{A}$ that still outperforms a_1 on all problems in P_1 completely changes this picture, provided that a functioning algorithm selection mechanism is in place. Now, the overall performance of the simulation system *decreases* if a_1 is still selected for

[1] UML stands for *Unified Modeling Language*, which is widely used to model (object-oriented) software. See [86] for a brief introduction to the original standard. As UML will be solely used here to *communicate* design aspects, I follow Fowler's advice [86, p. 6–7] and 'bend' the standard notation occasionally, in order to clarify the presented ideas (e.g., by the use of color or line styles).

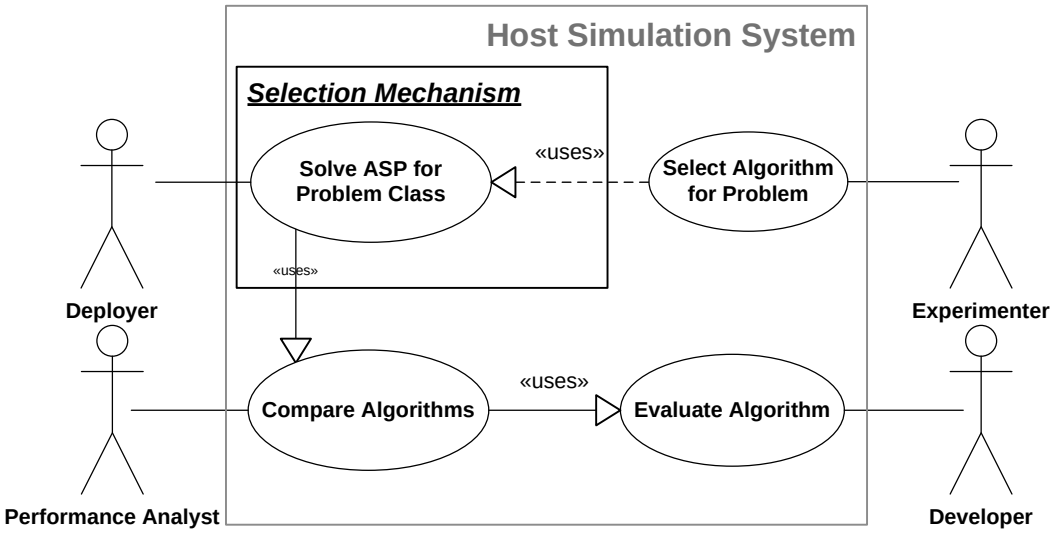

Figure 4.1.: General use cases related to simulation algorithm selection. In the end, the selection mechanism shall deliver automatic configuration of the host simulation system as requested by the system's end users, who conduct simulation experiments (experimenter role, upper right corner). In turn, empirical selection needs access to the host simulation system in a way that allows to *reason* about different algorithms and to *evaluate* them (semi-) automatically; a requirement that is also relevant for performance studies (performance analyst role, lower left corner) and iterative simulator development (developer role, lower right corner). However, the selection mechanism also introduces a new responsibility: before the deployment of a simulation tool, the automatic selection mechanism has to be configured accordingly (deployer role, upper left corner). The dashed *uses*-arrow at the top shall highlight the fact that the selection task *does* require solving the ASP, but that this is not required to happen at the same time (see sec. 2.6, p. 47).

problems in P_2 (where it now performs worse) but not for problems in P_1 (where a_2 is still faster). Such trade-offs may be strongly dependent on implementation details; to identify them is non-trivial and requires careful experimentation.

In [235], Moret describes a software development process that reiterates algorithmic changes and subsequent experimentation to achieve good performance, leading to the new discipline of *algorithm engineering*. For developers engineering their algorithms, an algorithm selection mechanism could provide valuable feedback regarding the relevance of the algorithm they work on, and also regarding the problem domain in which it excels. In the context of a flexible plug-in based system like JAMES II, such algorithms are not necessarily monolithic simulation algorithms, but could be random number generators, event queues, partitioning algorithms, and so on (see sec. 4.2, p. 81). The more available options there are, the harder is it to assess the *overall* impact of changing an implementation, since many algorithm *combinations* now need to be evaluated.

On a more general level, not only algorithm developers may profit from the feedback by an algorithm selection mechanism, but also researchers that are concerned with comparative performance analyses. As knowledge on the performance differences between algorithms is essential for any prior selection, performance analysts require the same experimentation features from a simulation system that are also necessary for algorithm selection. In other words, both algorithm developer and performance analyst are interested in simulator performance, but on different levels: the developer is focused on a *single* (sub-)algorithm (e.g., in order to optimize it), whereas the performance analyst is interested in comparing *multiple* algorithms.

The automatic selection mechanism as such also introduces a new role. The selection mechanism has to be configured regarding the techniques to be used, the data to be analyzed, and so on. Since the mechanism's main goal is to facilitate the usage of the *overall* simulation system, this responsibility

is likely to be in the hands of the *deployer*, i.e., a developer that aims to roll out a version of the simulation system that is easy to use. Instead of using the selection mechanism as some kind of high-level feedback, as in case of development or performance analysis, the deployer is not interested *which* algorithm gets selected, as long as it is a suitable one. Hence, the deployer has to configure the selection mechanisms to be supported by the deployed version of the simulation system. Making this decision may involve the application of meta-learning algorithms on top of the simulation algorithm selectors. Furthermore, the overall deployment gets more complex, as it now includes the generated selection mappings. This is an aspect none of the other roles has to deal with.

Finally, note that the discussed use cases refer to different *roles*, not necessarily carried out by different *persons*. For example, after iteratively developing a better simulation algorithm for a specific class of problems (*developer* role), a researcher may compare the performance of the new algorithm to many other setups (*performance analyst*), and finally deploy an updated version of the system, containing the new algorithm *and* and a selection mechanism that is adapted accordingly (*deployer*).

4.2. Brief Introduction to JAMES II

JAMES II [303] is a general-purpose open-source framework for modeling and simulation, developed in Java 1.6. It aims at providing a solid foundation that integrates various methods for modeling and simulation, as well as some adjacent techniques, e.g., simulation-based optimization.[2] Most of JAMES II's architectural groundwork has been conceptualized and implemented by Himmelspach; it is further detailed in [131, 136]. This introduction focuses on those aspects of JAMES II that are relevant for algorithm selection. It discusses its properties from the viewpoint of section 2.4.3, namely as a (potentially) adaptive software system, and illustrates the limitations of the current version with respect to the algorithm selection problem.

4.2.1. Fundamentals

The Abstract Factory Pattern

One of the central goals of JAMES II is enabling the re-use and re-combination of functionality via a plug-in system called *Plug'n Simulate* [136]. It is based on the well-known *Abstract Factory pattern* [98, p. 87], which in turn is based on the notion of *Factory classes*. The pattern is used to hide the intricacies of object creation in object-oriented languages.

A factory class makes object creation dependent on the parameters handed over, i.e., it encapsulates the (potentially complicated) creation process of an object; upon completion, the newly created object is passed back to the caller. For example, consider the code in listing 4.1 on page 82 (l. 11): the method `create(int parameter)` returns an instance of `MyClass` if `parameter` is greater than 100, otherwise it returns an instance of `MyOtherClass`. This *selection* between different implementations (or configurations of the same class) is done within the factory, and is hence invisible to the caller. By doing so, three different concerns can be separated from each other in a concise manner: the desired functionality (implemented by `MyClass` and `MyOtherClass`), the creation of objects that are capable of executing the desired functionality (implemented in `MyFactory`), and finally the usage of the functionality, as shown in the `main` method of listing 4.1 (l. 17). A corresponding UML class diagram is shown in figure 4.2.

From an algorithm selection viewpoint, factories can be regarded as a standard method to implement an explicit, hard-coded selection mapping in an object-oriented programming language. The separation of the aforementioned concerns (implementation, instantiation, and usage), be it via a dedicated factory class or not, is also regarded as a hallmark of component-based software systems (see sec. 2.4.3, p. 36).

[2] JAMES II is still under development. Its code base changes on a daily basis, so a discussion of concrete software entities may become outdated (e.g., w.r.t. package names). The version discussed here is revision 17.000 (April 14, 2010).

```
1  interface MyInterface {
     void doSomething();
3  }
   class MyClass implements MyInterface {
5    void doSomething() {...}
   }
7  class MyOtherClass implements MyInterface {
     void doSomething() {...}
9  }
   class MyFactory {
11   MyInterface create(int parameter) {
       //Create either MyClass instance or MyOtherClass instance
13     return parameter > 100 ? new MyClass() : new MyOtherClass();
     }
15 }
   //..
17 public static void main(String[] args) {
     MyFactory myFactory = new MyFactory();
19   MyInterface myObject = myFactory.create(someCalculation());
     myObject.doSomething();
21 }
```

Listing 4.1: Using a factory class: when `myFactory.create(int p)` is called, the user code in the main method does not predetermine which implementation of `MyInterface` is used — this is done in `MyFactory` instead.

While a factory class succeeds in separating these concerns, it fails to scale with the amount and diversity of possible implementations. Imagine there are many implementations of `MyInterface` (see listing 4.1), each with its own implementation-specific creation mechanism. Integrating all these mechanisms into a *single* factory would not only violate good coding practice (data hiding, separation of concerns), it would also require to change the factory every time a new implementation is added to the system.

This problem is alleviated by the Abstract Factory pattern. The pattern basically consists of factories that are nested twice, to separate the concern of *selecting* the implementation from the concern of *instantiating* the selected implementation. A so-called *concrete factory* is provided for each implementation. It encapsulates the code to instantiate a certain class. All concrete factories have a common superclass that declares the method for instantiation (see fig. 4.2). Such a super class is called *base factory* in JAMES II.

On top of this, an *abstract factory*[3] manages the concrete factories and is responsible for their selection, i.e., it chooses one of the concrete factories and passes it back. The returned concrete factory is a sub-class of the base factory, and can hence be called to create some instance that implements the desired interface. An exemplary implementation of the Abstract Factory pattern is shown in listing 4.2 (p. 84). Here, both `MyClass` and `MyOtherClass` have their own (concrete) factory, `MyClassFactory` and `MyOtherClassFactory`, respectively. Both are subclasses of `MyBaseFactory` and *only* concerned with instantiating their 'own' implementation of `MyInterface`, i.e., both factories are independent from each other. The selection code that chooses between both implementations has moved from the single factory in the Factory pattern (cf. listing 4.1) to the abstract factory `MyAbstractFactory`. While this separates the selection from the instantiation code, it also makes object creation a little more complicated, as exemplified in the `main` method of listing 4.2 (l. 21): object creation now involves the creation of an abstract factory, letting it create a concrete factory, and to finally let the concrete

[3] An abstract factory is *not* abstract in the sense of Java, i.e., it does not contain abstract methods and can thus be instantiated.

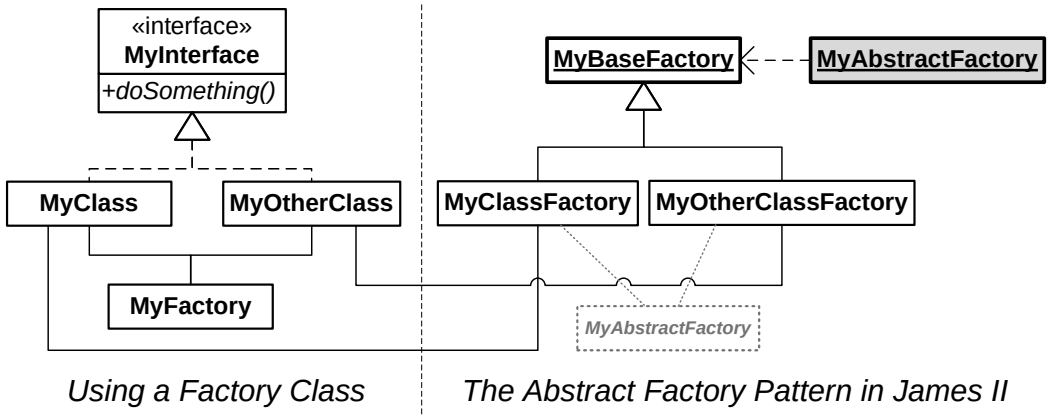

Figure 4.2.: A UML diagram to outline the use of factories (left) and the Abstract Factory pattern (right). The abstract factory pattern as realized in JAMES II lets `MyAbstractFactory` only rely on a base factory. The base factory has to define all operations that the abstract factory may need to decide on a conrete factory. This avoids to know the concrete factories `MyClassFactory` and `MyOtherClassFactory` at compile time, as sketched gray and dashed on the lower right. The notions of base and abstract factory (bold, underlined) are essential for the flexibility of JAMES II.

factory create an object implementing the desired functionality.

What has been won by separating instantiation and selection with the Abstract Factory pattern? The algorithm selection is still hard-coded, it just has been moved to a new dedicated class — the abstract factory, which now seems to depend on all concrete factories. However, this impression is deceptive: it is possible to let the abstract factory only depend on the base factory, given that the base factory declares all methods that are required by the abstract factory to make a proper selection (see fig. 4.2). Each concrete factory has to implement those methods. Furthermore, one could now find a way to dynamically detect all available concrete factories at runtime, and make them visible to the abstract factory. If these preconditions are fulfilled, the abstract factory is able to make a selection *without* relying on *any* concrete factories, in contrast to the simple implementation shown in listing 4.2. Since the Abstract Factory pattern allows to make a suitable choice from an unknown number of concrete factories, which might even be discovered at runtime, it can serve as a building block for a plug-in system.

Plug-ins and Plug-in Types

JAMES II is designed as a *framework* for modeling and simulation. A framework can be regarded as a collection of reusable functions that is complemented by predefined 'flows of control' [51, p. 948]. Frameworks should ease the development of more specialized applications on top of them and hence need to be extensible. JAMES II is extensible with respect to the entities used in the control flow and also by the *kinds* of entities it works with. Each type of entity corresponds to a so-called *plug-in type*. A plug-in type is represented by a specific implementation of the Abstract Factory pattern, which is based on some principal JAMES II classes. Since all dependencies on concrete factories shall be avoided, a plug-in type definition basically consists of an abstract factory and a base factory.

A concrete factory and the functionality it provides are regarded as a *plug-in* [131, p. 59–63]. The concrete factory is responsible for instantiating a certain implementation of an interface or an abstract class, corresponding to the factory's plug-in type.[4] A plug-in can be regarded as a software *component* in the sense of section 2.4.3 (p. 36), as it is separated from the rest of the system (by its concrete factory) and can be developed and deployed independently — two properties that are regarded as the main characteristics of software components (e.g., [230, p. 58]). Plug-in components

[4] More specifically, a factory implements the `create(ParameterBlock params)` method of the plug-in type's base factory.

```
class MyBaseFactory {
  abstract MyInterface create(int parameter);
}
class MyClassFactory extends MyBaseFactory {
  MyInterface create(int parameter) {
    //Create MyClass instance, depending on passed parameter:
    return parameter > 100 ? new MyClass(true,1,3.1415) : new MyClass(false);
  }
}
class MyOtherClassFactory extends MyBaseFactory {
  MyInterface create(int parameter) {
    return MyOtherClass(true, parameter); //Create MyOtherClass instance
  }
}
class MyAbstractFactory {
  MyBaseFactory create(int parameter) {
  return parameter > 100 ? new MyClassFactory() : new MyOtherClassFactory();
  }
}
//..
public static void main(String[] args) {
  MyAbstractFactory myAbstractFactory = new MyAbstractFactory();
  MyBaseFactory myFactory = myAbstractFactory.create(someCalculation());
  MyInterface myObject = myFactory.create(someCalculation());
  myObject.doSomething();
}
```

Listing 4.2: Using the Abstract Factory pattern. `MyInterface`, `MyClass`, and `MyOtherClass` are defined as in listing 4.1. When `myAbstractFactory.create(...)` is called, it creates the concrete factory to be used, which is used to instantiate an implementation of `MyInterface`. Note that the code in `main` is independent of the implementation, as it merely calls a method from the common super class of all concrete factories, i.e., from the base factory `MyBaseFactory`.

are the fundamental element of reuse and composition in JAMES II [131, 136]. Each offers a specific functionality, so that they can serve as building blocks for the simulation algorithms that shall be selected automatically. In other words, the algorithm set $\mathbb{A}$ of the ASP (def. 2.1.1, p. 14) is constructed by considering *combinations* of plug-in components. Definitions for both plug-in types and plug-ins are stored in *XML (eXtensible Markup Language)* files named `plugintype.xml` and `plugin.xml`,[5] respectively. Sample definition files for plug-ins and plug-in types can be found in listings A.1 and A.2 (p. 228).

The JAMES II Registry

All abstract and concrete factories are assumed to have empty constructors, so that they can be instantiated dynamically at runtime. This is implemented by the JAMES II *registry* during start-up. The registry scans a specific directory for all plug-in (type) declaration files. Then, it instantiates the factories, stores the instances, and assigns all concrete factory instances of one plug-in type to the corresponding abstract factory instance.

The registry itself is implemented as a *singleton* [98, p. 127], i.e., there is only one instance of it per *Java Virtual Machine (JVM)*. It can be retrieved by calling `SimSystem.getRegistry()`. In order to

[5] The name `plugin.xml` is somewhat ambiguous, as each file may contain more than one concrete factory of the same plug-in type. In fact, a JAMES II plug-in may consist of *multiple* factories according to [136, p. 2], while the term seems to refer to a single factory (and its parameters) in [131, p. 63]. In the following, the term plug-in will only be used to refer to a *single* concrete factory and the implementation it provides.

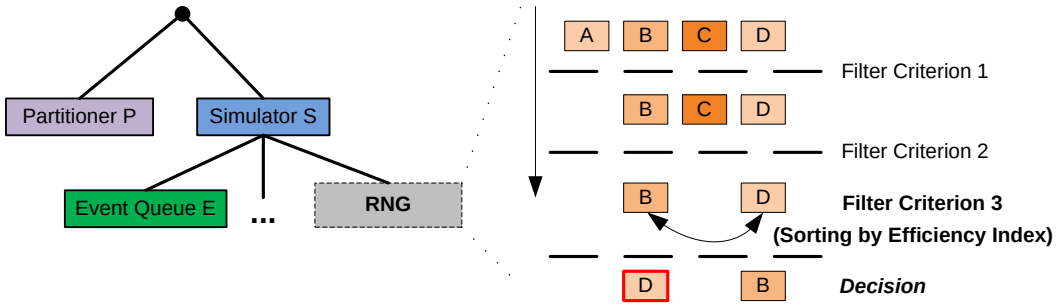

Figure 4.3.: Outline of the factory filtering process in JAMES II. The tree structure on the left sketches the dependencies between all plug-ins used in the given example, whereas the right side shows the factory filtering process to select an RNG. Here, a partitioning algorithm P has already been selected, as well as a simulation algorithm S and an event queue implementation E. The selection of a suitable RNG plug-in from the options $A, \ldots, D$ (upper right corner) involves the subsequent application of *filtering criteria*. For some plug-in types, an additional criterion *re-orders* the options that are left (e.g., sorting by a so-called efficiency index). Since the first factory from the list (red box) is finally selected by the registry, this enables simple algorithm selection mechanisms.

use a JAMES II plug-in for a certain task, the registry is called by

```
SimSystem.getRegistry().getFactory(MyAbstractFactory.class, parameters);
```

where the first argument specifies the plug-in type of interest (by passing the class of the corresponding abstract factory), and the second argument is an instance of `ParameterBlock`. `ParameterBlock` is an auxiliary class that represents a tree structure containing arbitrary objects at each node. It is used to pass along structured and complex parameters system-wide in a standardized manner.

Factory Filtering

When the `getFactory(...)` method of the JAMES II registry is called, the registry propagates the call to the corresponding abstract factory, which returns a list of all *eligible* concrete factories, i.e., all factories that are able to fulfill the task specified by the passed parameters. The eligible factories are identified by applying a sequence of *filter criteria*, as shown in figure 4.3. Then, the registry checks if the list is empty, in which case there is no entity that can solve the current problem and an error is reported. If the list is non-empty, the *first* element from the list of concrete factories is passed back to the caller.

This straightforward mechanism makes it possible to integrate simple algorithm selection schemes by adding a filter criterion that does not actually *filter out* any factory, but rather *sorts* them. Such criteria had already been implemented for simulation algorithms and event queues, to solve the ASP in an ad-hoc manner. The base factory is extended by an additional function, `getEfficiencyIndex()`, which returns a floating-point number that describes how efficient the developer deems an implementation to be *in general*.[6] The factory list is then re-sorted in descending order of the efficiency indices given by the concrete factories.

Alternatively, one may also implement an application-specific algorithm selection within the user code, by calling the registry's `getFactoryList(...)` method. It returns the whole list of eligible factories, from which the application programmer's code can then select one arbitrarily. This is particularly useful for orchestrating the interplay between different components, e.g., if one component relies on a data structure produced by another one and the type of the data structure is not known at compile time. This is the case for the partitioning sub-system of JAMES II, which combines three types of components: one for constructing a model graph from the current model's structure, one for

[6] Note that `getEfficiencyIndex()` does not rely on any parameters, so it is independent of any problem features.

constructing an infrastructure graph of available processors and their network topology, and one for mapping the model graph onto the infrastructure graph, i.e., the actual partitioning step [69, p. 849]. Each of the three tasks corresponds to a single plug-in type. All three interfaces provided by the plug-in types are defined on graphs, but these graphs can be annotated by additional information — on which a partitioning algorithm may or may not rely. This makes it necessary to find a suitable *combination* of components at runtime, which is implemented most conveniently by querying the registry for all eligible factories per plug-in type, and then applying a customized selection procedure. The need for searching such combinations does not occur very often, because most interfaces in JAMES II are defined in a way that allows their free combination: subsequent calls to `SimSystem.getRegistry().getFactory(...)` suffice to find a suitable set-up. However, even if such a customized selection procedure has to be implemented, it can still be kept consistent with the semantics of the factory lists returned by the registry, just by favoring factories that appear at the *beginning* of the lists retrieved from the registry.

Note that the JAMES II registry can be called from anywhere in the system, so after selecting a concrete factory to create a desired component, the selected factory may itself call the registry to retrieve *other* factories for the instantiation of auxiliary components. For example, the partitioning system mentioned above also contains *multi-level partitioning* partitioning schemes as described in [1], which rely on additional algorithms to coarsen and refine the model graph.

Further Design Aspects

As the previous sub-sections already suggest, one of the major goals of JAMES II is to provide maximal flexibility to anyone who develops modeling and simulation applications on top of it. This goal is also reflected in other aspects of the overall architecture:

- *Independence from external code*: the integration of other software, e.g., external libraries, can be very challenging (e.g., see discussion in [40]) and makes software deployment, installation, and maintenance more difficult. Therefore, the core of JAMES II is independent of any external software.[7] This does not mean that external libraries should not be used at all — their use is just restricted to plug-ins that are separated from the core functionalities, and they should be accessed through a *wrapper* (see listing A.3, p. 229).

- *Interfaces to avoid forced inheritance*: All essential classes in JAMES II are complemented by interface definitions. While one may still inherit from the standard implementation to alter the functionality in a simple manner, it is also possible to develop a new implementation of the interface from scratch, e.g., an implementation that relies on external code and therefore has to inherit from a specific super class. Most general methods in JAMES II only operate on interfaces. The interfaces form a hierarchy much similar to that of the default classes that implement them.

- *Parameter block nesting:* The aforementioned `ParameterBlock` instances are used to pass all required information to both abstract and concrete factories. In order to create auxiliary components, any concrete factory may call the registry by itself. Hence, it may require additional parameters for its sub-components. Parameter blocks can therefore be nested arbitrarily, so that they are able to set up a whole hierarchy of components. Each factory receives parameters for their sub-components via their own parameters, and has to propagate the sub-parameters accordingly (without having to consider their specific content). This mechanism plays a key role for implementing automatic algorithm selection in JAMES II (see ch. 8).

- *Patterns:* Besides the omnipresent Abstract Factory pattern, several other design patterns have been used to cope with specific issues. Most JAMES II simulators are based upon the Template pattern [98, p. 325], to avoid redundancy and make their code re-usable. For example, the hierarchy of sequential and parallel simulation algorithms presented in [133] relies on this

[7] Except for the Java Virtual Machine it is executed on, obviously.

pattern. Similarly, the Observer pattern [98, p. 293] is employed for data collection. Advanced architectural patterns make the distribution of simulation runs more flexible [201].

4.2.2. Relation to Self-Adaptive Software

Software-centric approaches that deal with the algorithm selection problem are focused on constructing *adaptive* software systems (see sec. 2.4.3, p. 36). Features of such systems are therefore of particular interest, as they could hint at elements that are still missing in JAMES II but that are necessary for adaptivity in terms of algorithm selection. McKinley et al. regard component-based design, the separation of concerns, and reflection as *"enabling technologies"* [230, p. 57] for (compositional) adaption:

- *Component-based design* is implemented by the plug-in system of JAMES II. Each plug-in can be considered as a single component (sec. 4.2.1, p. 83). The way that components interact is predetermined by other entities within the framework. New components can be constructed from existing ones, and the actual selection of the components is already done automatically at runtime, albeit in a simplistic manner (see fig. 4.3, p. 85).

- *Separation of concerns* is required by McKinley at al. to separate *"business logic"* from *"cross-cutting concerns"* like robustness or security [230, p. 58]. This is only partly achieved in JAMES II: while there are separate sub-systems for many cross-cutting concerns of modeling and simulation — e.g., distributed execution, data collection, or random number generation — more general concerns have been tackled in many different ways. A few issues are managed by dedicated entities (e.g., error reporting), some are partially supported but also intertwined with application code (e.g., robustness), and some have been neglected altogether so far (e.g., security). Since the automated selection of simulation algorithms can be regarded as another concern, it should be separated from the rest of the system.

- *Reflection* can be considered with respect to behavior or structure of a software system. Structural reflection in JAMES II is possible to some extent, e.g., via querying the registry about the plug-ins that are loaded or by using the reflection API of Java [245]. However, as discussed in section 2.4.3, *behavioral reflection* is more relevant for algorithm selection, as the system shall select algorithms by considering their past performance, i.e., their behavior. This is *not* inherently supported by JAMES II. McKinley et al. further subdivide reflection into two distinct activities: *introspection*, i.e., the observation of own behavior, and *intercession*, i.e., the modification of the system's state in order to *reflect* those observations. To support the degree of behavioral reflection that is required for algorithm selection, JAMES II has to allow the observation of algorithm performance (i.e., introspection) and the accordant adjustment of the selection mapping it implements (i.e., intercession).

4.2.3. Limitations of Algorithm Selection in JAMES II

Apart from the missing capabilities for behavioral reflection (see sec. 4.2.2), the basic plug-in system of JAMES II also imposes some other limitations on implementing an algorithm selection mechanism.

Clearly, the current workaround to avoid the algorithm selection problem — i.e., sorting the factories by a hard-coded efficiency index[8] as described on page 85 — is not flexible enough to express arbitrary selection mappings. As already elaborated in section 2.1.2 (p. 16), such total orders of algorithms (where one algorithm dominates all others, and so on), are rather unlikely, because the performance of most algorithms depends on the properties of the problem instance they are applied to. Another

[8] A generalized form of this scheme, which allows to also consider the passed parameter block, was implemented for prototyping potential ASP solutions. It works on the interface `james.core.factories.IParameterFilterFactory` and is used by many plug-in types in JAMES II. However, similar limitations as for the original efficiency index solution apply.

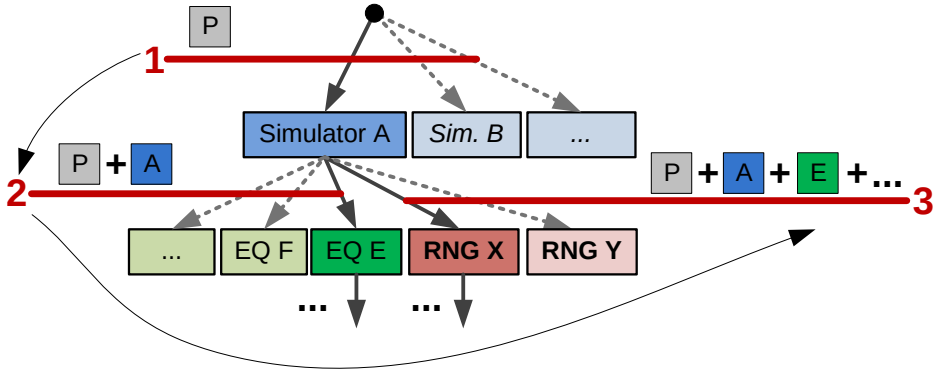

Figure 4.4.: Missing context for ad-hoc algorithm selection. In the above figure, the ASP has to be partially solved (at least) three times, denoted by the three red bars. Each bar represents the invocation of the registry, which in turn triggers the factory filtering process (cf. fig. 4.3, p. 85) for the given plug-in type, in order to select a suitable implementation. Each time, the context to be considered by the selection procedure gets more complex: the simulation algorithm A is chosen by only considering the simulation problem $P \in \mathbb{P}$, while the event queue E is chosen by considering the simulation problem P *and* the kind of simulator that was already selected (A, see step 2), and so on.

problem is that a *comparison* by efficiency index implies that all indices have the *same* scale and refer to the *same* metric. This also means that all indices depend on each other, which violates their strict separation and independence (see sec. 4.2.1). Apart from the difficulties of comparing algorithms in general (see sec. 3.1, p. 61), the goal of finding a *single* number that expresses *all* performance aspects of a plug-in component for *all* relevant problem instances and that is consistent with the indices of *all* other plug-ins is unrealistic. Hard-coded efficiency indices also hamper the *robustness* of the plug-in system, i.e., its ability to cope with malfunctioning or even malicious plug-ins. The factory shown in listing A.4 (p. 229) exploits the sorting by efficiency index: it always returns the maximal index that is possible (i.e., `Double.MAX_VALUE`). If such a plug-in would be deployed, it would prevent the automatic selection of *any* other simulation algorithm. For the same reasons, sorting efficiency indices does not cope well with malfunctioning plug-ins. These will be selected again and again, until the user manually intervenes by either re-compiling the buggy plug-in with a lower efficiency index, removing its plug-in file, or explicitly specifying an alternative plug-in.[9]

Furthermore, factories are selected by the registry in an ad-hoc manner, without a simple way to access the *overall* problem context. For example, consider the concrete factory of a simulation algorithm A, which calls the registry to provide it with a suitable event queue, as shown in figure 4.4. The abstract event queue factory is not able to consider its context, i.e., it is not informed that the selected event queue shall be used within A (instead of simulator B, for example) — unless this context information is *explicitly* stored in the parameter block that is passed to the abstract event queue factory. Clearly, the required context could be much more complex and would potentially include the whole simulation problem! This would have to be propagated to all abstract factories that are involved in the creation process. Thereby the context gets larger and larger, since it also has to specify which algorithms have already been selected (and with which parameters), i.e., a partial solution of the current ASP. In figure 4.4, the simulation algorithm and its set-up would be the partial ASP solution that needs to be considered by the abstract event queue factory. Hence, even if there is a suitable mechanism for propagating the current context, the selection of the *whole* simulation algorithm — i.e., a specific combination of plug-in components — would be distributed over *all* abstract factories that are involved. Developing filter criteria for such a distributed algorithm selection is quite challenging: at least one filter criterion of each abstract factory has to be aware of *all* relevant contexts in which the corresponding abstract factory has to make a choice. If, for example, a

[9] A simple technique to alleviate this problem is presented in section 8.1.2 (p. 162).

new simulation algorithm is introduced and it requires an event queue, the factory criterion for event queue selection would have to be updated. It has to account for the new potential context. All this introduces additional complexity, as the 'global' selection mapping to be determined in definition 2.1.1 (p. 14) is now replaced by a conjunction of many 'local' selection mappings, each having a different domain (depending on the context).

4.3. Technical Requirements for Algorithm Selection in JAMES II

Some of the major technical requirements for a simulation algorithm selection mechanism have already been discussed in section 4.2.2 (p. 87): algorithm selection can be regarded as a cross-cutting concern, because it relates to many plug-in types in JAMES II. It should therefore be implemented *separately* from the current registry's factory selection mechanism. Otherwise, it would also have to overcome the practical limitations of JAMES II's current plug-in system, as discussed in section 4.2.3. Furthermore, any sophisticated automated algorithm selection method relies on behavioral reflection, i.e., JAMES II has to provide techniques for introspection (i.e., observing performance) and intercession (i.e., updating the selection mapping accordingly). The specific requirements arising from all these aspects are detailed in the following.

Scalability

Before specifying concrete functional requirements, it should be clear under which circumstances the ASP has to be tackled for JAMES II. Clearly, all techniques need to scale with the amount of empirical performance observations from which to inductively derive selection mappings, and also with the number of eligible algorithms in $\mathbb{A}$. While the amount of available data grows linearly with the number of simulation runs executed for performance analysis, the set of algorithms may grow *exponentially* with the number of new implementations. This is due to the plug-in system of JAMES II, which allows additional extensions in form of plug-in types, as well as a flexible *combination* of existing algorithms to solve a given simulation problem. These factors may lead to a combinatorial explosion.

More formally, let $\mathcal{P}$ be the set of all JAMES II plug-ins, partitioned by n plug-in types into pairwise disjunct sets $P_1, \ldots, P_n$, i.e., $\bigcup_{i=1}^{n} P_i = \mathcal{P}$ and $\forall P_i, P_j : P_i \cap P_j = \emptyset \iff i \neq j$. Defining a new plug-in type means to add another set P_{n+1} that is initially empty, whereas defining a new plug-in of type i means to add an element to the set P_i. In the ideal case of freely exchangeable plug-ins, a simulation algorithm relying on two auxiliary plug-ins of type x and y can be configured in $|P_x| \cdot |P_y|$ ways. Adding a new plug-in of type x or y will hence result in a polynomial growth of potential algorithm combinations.

However, a developer may also add a new plug-in *type* z, e.g., for separating a part of the simulator's implementation that might be useful in other contexts, has a clear interface, and could be solved in several ways that are non-trivial to compare. Even if merely two alternatives are implemented for the new plug-in type z, i.e., $|P_z| = 2$, this *doubles* the number of possible combinations, which is now $|P_x| \cdot |P_y| \cdot |P_z|$. The definition of new plug-in types could hence result in a combinatorial explosion. The increasing numbers of both plug-in types and plug-ins in JAMES II have already been illustrated in the introduction (see fig. 1.2, p. 3). For some classes of simulation problems, e.g., chemical reaction networks (see sec. 1.3.1, p. 4), JAMES II already offers *hundreds* of feasible algorithm combinations. Each new event queue or simulator implementation further complicates the creation of selection mappings by increasing the number of possibilities. It is therefore important that the algorithm selection mechanism is able to deal with large numbers of algorithms. This might be done by considering the particular structure of the algorithm set $\mathbb{A}$ in JAMES II. In other words, an algorithm selection mechanism for JAMES II has to cope with algorithms that are neither monolithic nor just parameterizable, but rather *combinations* of plug-ins interacting with each other (see sec. 2.6.1,

p. 49).

Introspection

Supporting introspection means that it has to be possible from within JAMES II to observe the performance of plug-ins or plug-in combinations. This feature can be decomposed into three functions that have to be implemented:

1. Recording the performance of a single simulation run.
2. Storing the performance data.
3. Retrieving the performance data associated with a certain algorithm and simulation problem.

Recording the performance of a simulation run should involve minimal user interaction. The impact of confounding factors needs to be minimized, in order to avoid the introduction of a systematic bias to the results. In other words, performance measuring needs to be as little intrusive as possible. Then, the recorded performance data needs to be stored for latter retrieval. Given the aforementioned scalability issues, the storage system should be able to cope with large amounts of data, considering that each eligible algorithm combination might be applied to multiple simulation problems. The performance data has to be retrievable by various characteristics, e.g., by simulation problem and by the specific algorithm (i.e., plug-in combination). Furthermore, the data storage has to take into account the hardware that was used to collect the performance data, to ensure its reproducibility.

Finally, introspection should be enhanced by some of the experiment design techniques mentioned in section 3.2, which improve experiments regarding their statistical efficiency and the insights that can be gained from their results. This could lead the way to automated approaches of algorithm performance evaluation — another important requirement, as one cannot expect all algorithm developers to be familiar with experiment design.

Intercession

Algorithm selection can be regarded as a technique to enable the intercession of an adaptive software system. A simulation system may adapt its behavior by altering the *selection mapping* (see sec. 2.1, p. 13) that decides on the algorithm to use.

As chapter 2 illustrates, there are many fundamentally different ways to come up with a selection mapping, e.g., machine learning (sec. 2.3, p. 24) or general adaptation schemes like genetic algorithms (sec. 2.4.2, p. 35). Furthermore, each field gave rise to various techniques, all with their advantages and disadvantages. The inevitable compromise that comes with the choice of a specific selection mapping generation method is most evident in machine learning, where the optimal trade-off between bias and variance (see sec. 2.3.1, p. 25) depends on the given data: there is *no* silver-bullet learning algorithm. Instead, the best one needs to be chosen by considering the features of the problem at hand. Clearly, selecting a machine-learning *algorithm* to automatically generate selection mappings requires to solve the ASP on yet *another* level of abstraction. This level of abstraction is addressed by meta-learning (see sec. 2.7, p. 55 and fig. 2.15, p. 59). Although meta-learning as such is not in the focus of this discussion, the algorithm selection mechanism should *enable* future meta-learning schemes by providing means to *generate*, *evaluate*, and *manage* selection mappings with *different* methods. New selection mapping generation methods may be implemented at later times, so that there should be a mechanism to plug them into an existing setup of JAMES II.

While the implementation of automatic algorithm selection should also be as transparent to users as possible, i.e., all automatable parts of the process should be hidden, users still need to exert full control over the system. More specifically, they should be able to decide whether or not to use the automated algorithm selection mechanism at all. Several reasons may *prevent* a user from relying on automatic algorithm selection, depending on the given situation:

- No automatic selection is necessary, because the user knows the best algorithm (or there is only one).
- The performance of a *specific* algorithm shall be observed.
- The automatic selection mechanism does not yet consider the performance measure of interest.
- The automatic selection mechanism is known to be ineffective for the given problem.

Therefore, automatic algorithm selection should be easy to switch on or off. This could even be done automatically in some cases, e.g., if the evaluation of selection mappings for a certain problem class already showed their ineffectiveness (see sec. 2.1.2, p. 16).

Finally, there are situations in which even a sub-optimal selection is much preferable over having no automated selection at all: in case of a failure, e.g., if an algorithm is not able to correctly process a model with certain specifics, an automatic selection mechanism could offer a *graceful* way to cope with the situation. It should select a fall-back component to replace the faulty plug-in and repeat the computation. All this, however, is only applicable if failures are *detected*, i.e., in Java such a mechanism would have to react on exceptions thrown at runtime. Clearly, algorithm selection cannot prevent all implementation errors in simulation algorithms.

Interaction with the Host Simulation System

In principle, a simulation algorithm selection mechanism could be implemented independent of any concrete simulation system, e.g., by simply defining a generic input format for all required data. Yet, only a certain level of *integration* with a host system allows to *automate* the selection process for users of the host system: the host system has to query the selection mechanism in case of a simulation execution. To still retain the generality of a system-independent selection mechanism, host system and selection mechanism should be separated as much as possible and only interact via clearly defined interfaces. It allows to re-use the selection mechanism across several simulation systems. This could be useful to compare algorithms across different systems, or to even transfer some high-level knowledge (e.g., on the accuracy of a shared library) from one system to the other.

To some extent, using JAMES II as a host system can be considered a 'worst case scenario' for simulation algorithm selection, due to the wealth of available algorithms that result from its flexibility. Hence, it seems reasonable to implement and evaluate prototypical selection mechanisms on top of it. This also makes it possible to assess the scalability of the implemented algorithm selection approaches, depending on the number of available algorithms and so on. Apart from providing numerous test cases, JAMES II also offers many useful solutions to tackle other requirements mentioned above, e.g., its general plug-in system. By restricting interaction between JAMES II and the selection mechanism to specific interfaces, it is still possible to use the JAMES II-based prototype to select algorithms for other simulation systems, as long as they implement the same JAMES II interfaces. The loss of generality that stems from focusing on the development of an algorithm selection mechanism for JAMES II can thus be considered negligible.

Summary

Table 4.1 summarizes the major technical requirements for the simulation algorithm selection framework (SASF) to be developed for the host system JAMES II. Clearly, keeping them separated applies to both systems (first line), as well as their scalability with respect to the number of available algorithms. This task, however, is much easier to accomplish for the simulation system: while the host simulation system merely has to *manage* the algorithms and to specify how they can be combined with each other, the algorithm selection problem to be solved by the SASF is likely to get *harder* the larger $\mathbb{A}$ is.

Behavioral introspection, i.e., the ability to automatically evaluate and compare existing parts of the simulation system, could be implemented either by the selection mechanism or by the host

Requirement	SASF or Simulation System?	JAMES II
Separation of Concerns	both	✔
Scalability w.r.t. #Algorithms	both	✔
(Automated) Behavioral Introspection	any/SASF	✖
Scalability w.r.t. #Performance Data	SASF	-
Behavioral Intercession	SASF	-
Meta-Learning / Performance Evaluation	SASF	-

Table 4.1.: Summary of the identified requirements. Checkmarks denote the features that are already available in JAMES II, crosses denote missing features, and hyphens denote cases that are of no concern for JAMES II.

system (or by both). It is an important aspect of any empirical approach to algorithm selection. The empirical performance data provided by behavioral introspection are then processed to generate selection mappings. Generation, management, and evaluation of selection mappings is clearly in the responsibility of the selection mechanism, so it has to scale with the amount of available performance data. The generated selection mappings should steer the behavioral intercession triggered by the selection mechanism. Finally, those selection mappings have to be evaluated and selected as well (meta-learning, last line).

4.4. A Simulation Algorithm Selection Framework

Given the analysis in the preceding sections, it seems reasonable to investigate the benefits and limitations of automated simulation algorithm selection by implementing a prototype for the modeling and simulation framework JAMES II. Since the prototype needs to include *several* realizations for certain tasks, such as the generation of selection mappings (see sec. 4.3, p. 89), the general and flexible plug-in system of JAMES II can be re-used here to ease implementation. It provides a solid and well-tested foundation for implementing any task related to modeling and simulation.

Although most concrete implementations of functionality shall be exchangeable, the prototype should not just provide a set of loosely coupled helper functions for realizing selection mechanisms. Instead, it should prescribe how the different techniques interact with the host system, thereby hiding the internal complexity of the overall task as much as possible. In other words, the major constituents of the selection mechanism need to be realized as a *software framework*: a *simulation algorithm selection framework (SASF)*. The idea of a general framework for simulation algorithm selection blends in nicely with the overall idea of JAMES II, namely to provide a *general* framework for any task related to modeling and simulation. This includes simulation algorithm selection.

Note that developing on top of JAMES II does *not* reduce the general applicability of the resulting simulation algorithm selection framework for *other* simulation systems, as long as they are able to interact with the SASF via its predefined interfaces. Nevertheless, the focus on JAMES II motivates its usage as an exemplary host system, e.g., to explore potential benefits that stem from integrating host system and selection mechanism.

While this section introduces the general architecture of the framework and discusses which elements are responsible for fulfilling which requirement, their specific design and functionality is detailed over the next chapters (ch. 5 to ch. 8). Before, the overall architecture of related projects will be briefly reviewed, which allows to identify major modules already identified by others, and hence puts the approach pursued here into an more general perspective.

4.4.1. Related Software Systems

There are several tools that aim at features similar to those the SASF shall eventually provide. They are aimed at different (or more general) application domains (see sec. 2.7, p. 53), but may still help

to come up with a feasible overall layout. Since the SASF is a framework, both its overall *structure* and the way its components work together, i.e., their *interaction pattern*, need to be identified.

Structure and Major Components

A rather elaborate software structure was realized for PYTHIA II [146], a problem solving environment that is focused on numerical solvers (see sec. 2.7, p. 53): the whole system is divided into different *layers* — user interface, data generation/analysis, and data storage — each of which consists of multiple components. The user interface layer is divided into two modules, one for the 'knowledge engineer' (similar to the *deployer* role, see fig. 4.1, p. 80) and one interface to use the system for algorithm selection. Users of the latter would have the role of experimenters (see fig. 4.1, p. 80). The second layer contains a single *"problem execution environment"* — in the SASF context, this would be an interface to the host simulation system. This layer also contains several modules for data analysis. The third layer is accessed via a single interface (of the database management system), but there are different kinds of data managed in different databases: input problems, performance data, problem features, and statistical data. PYTHIA II's overall layout is based on the principal ASP components already identified by Rice in [272] (cf. fig. 2.1, p. 14).[10] Its structure is summarized in the upper-left sketch of figure 4.5 (p. 94).

Dongarra and Eijkhout follow another direction in [60], where they introduce a *self-adaptive numerical software (SANS)* system whose central component is an 'intelligent agent' (see fig. 4.5, upper right sketch): it consists of a 'history database' to store information on past problems and past performance measurements, a 'system component' that allows to schedule tasks and interfaces the host system (e.g., a grid), and an 'intelligent component'. The intelligent component analyzes the data from the history database and is responsible for algorithm selection. The system is aimed at scientific computing in general, i.e., it includes the problem domains PYTHIA II was designed for. Instead of a graphical or web-based user interface like PYTHIA II, the SANS is accessed via some scripting language. Their approach is targeted towards programmers.

In [25], Brewer presents the structure of a *high-level library* as well as additional components to configure it (see fig. 4.5, lower left sketch). The high-level library consists of several algorithms for the same task (e.g., sorting). Each algorithm in the library is associated with a set of *models*, one for each platform that is supported. These are *statistical* models for performance prediction (in the sense of sec. 2.3.1, p. 25), not to be confused with simulation models. The models are created by an (external) 'auto-calibration toolkit'. A 'parameter optimizer' component searches for the best parameter setup per algorithm and platform, by considering an algorithm's models. The algorithm selection as such is done by the 'algorithm selector' component of the high level library, which queries the appropriate model for each algorithm and then selects the algorithm with best predicted performance. As with SANS, the high-level library is accessed by other programs only and does not require additional user interfaces.

On a more conceptual level, Karsai et al. [176] propose a structure motivated by adaptive control theory (see fig. 4.5, lower right sketch): to make a system adaptive, it can be extended by a 'supervisory layer' that contains a 'supervisory component'. The supervisory component is able to reconfigure the given system, which in this case would be the host simulation system. It is also able to monitor the performance of the components on the lower level. The observed data determines how the system gets adapted, e.g., this could be performance data from simulation algorithms. Observation is facilitated by an 'event monitor' that diagnoses the occurrence of certain events on the ground level and then triggers adaptation. Adaptation is implemented as a finite state machine that reacts on the generated events by initiating a reconfiguration.

To facilitate an overview, figure 4.5 compares the discussed architectures. PYTHIA II (upper left sketch) stands out in several ways: its architecture is organized in layers, it contains the most distinct modules, and instead of being invoked by other software it is accessed via user interfaces. Brewer's architecture for a high-level library (lower left sketch), in contrast, does not contain a module to

[10] This comes at no surprise, as Rice was involved in the construction of PYTHIA II and co-authored [146].

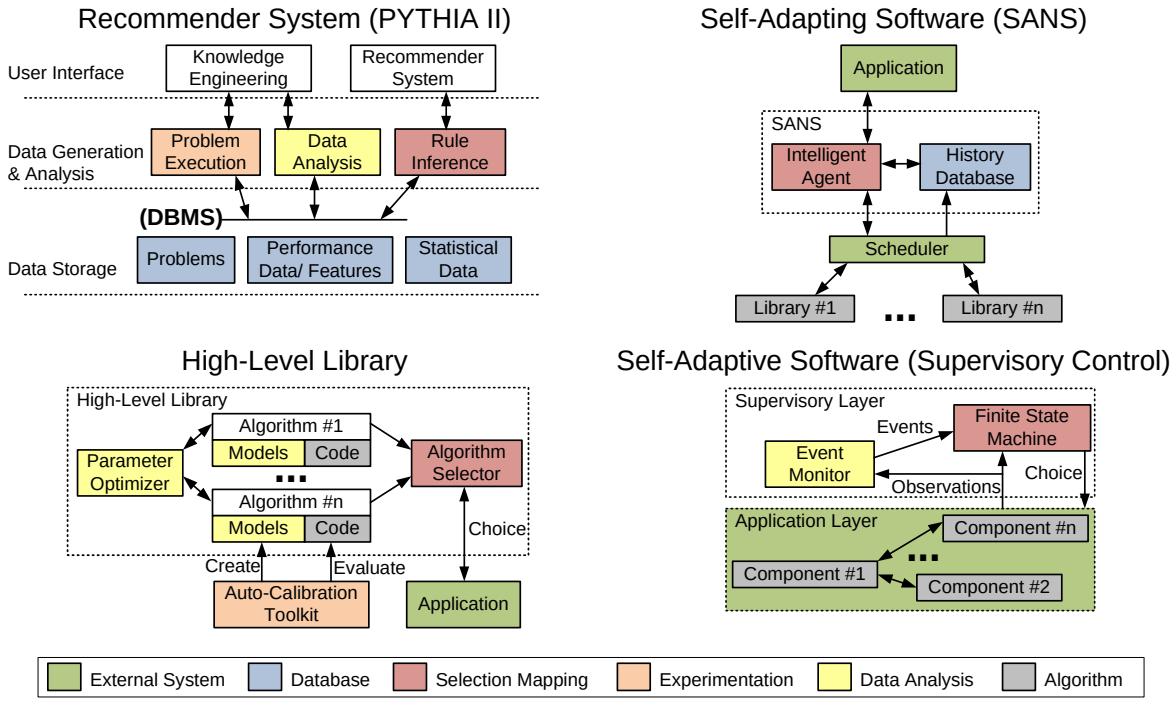

Figure 4.5.: Software architectures for algorithm selection. Note that some components have several functions and are merely colored by their most important functionality, e.g., the intelligent agent component in the SANS layout (upper right) does *also* analyze the data stored in the history database. The sketches have been adapted from [146, p. 234], [60, p. 126], [176, p. 30–32], and [25, p. 81] (clock-wise, starting with upper left).

store past performance data — only the *results* of analyzing the performance data, i.e., the statistical models associated with the algorithms, are stored. Both PYTHIA II and the high-level library approach have in common that they feature a dedicated component for experimentation (the problem execution module and the auto-calibration toolkit, respectively) — this is missing in the self-adaptive architectures on the right side of figure 4.5. The reason for this is simple: as these systems are supposed to adapt *at runtime*, they do not distinguish clearly between exploring the system's performance and selecting an algorithm — this functionality is included in their components for algorithm selection. An aspect all approaches have in common is the definition of a distinct component for algorithm selection that has a clear interface to the rest of the system.

In spite of that, Karsai et al. claim that *"adaptive control makes (at least) two important contributions to self-adaptive software: (1) the adaptation mechanism should be explicit and independent from the 'main' processing taking place in the system, and (2) the overall system dynamics should be different for the adaptation mechanism and the main processing mechanism."* [176, p. 26]. Nevertheless, both aspects can also be found in the other approaches that do not rely on adaptive control theory. Finally, note that — as already discussed in chapter 2 — the presented approaches originated in different research communities, and hence rely on differing terminology. The algorithm selection component, for instance, is implemented by a 'rule inference' module (upper left), an 'intelligent agent' (upper right), a 'finite state machine' (lower right), or an 'algorithm selector' (lower left). Partly, this divergence stems from the level of detail (e.g., code of single algorithms on the lower left, whole libraries on the upper right side) under consideration, but also from a lack of transfer between the different fields, as discussed in section 2.6 (p. 47).

Interaction Patterns

Besides architectures as such, it is also of interest which activities have been identified by other developers of algorithm selection systems, and how these activities are related to each other. How to map them onto the roles identified in section 4.1?

PYTHIA II offers a dedicated user interface for algorithm experts (see fig. 4.5, upper left sketch), which can be used to build a recommender system. In [146, p. 231], Houstis et al. name the following tasks to be done:

1. *Data Preparation*: This task involves the selection of suitable benchmarks (problem space $\mathbb{P}$), problem features (features space $\mathbb{F}$), performance metrics, and the execution of experiments with all relevant algorithms.

2. *Data Mining*: This task subsumes all automated analysis mechanisms applied to the data gathered in the preceding step. Houstis et al. consider it to be *"[...] the heart of the process"* [146, p. 231].

3. *Results Analysis*: This step relates to the *manual* interpretation of the data mining results, and should be carried out by 'knowledge engineers'.

4. *Knowledge Assimilation*: After approval and adjustment by the knowledge engineers, an inference engine (see fig. 4.5, upper left sketch) is used to generate recommendation rules. This has already been detailed in section 2.7 (p. 53).

PYTHIA II then allows end-users to specify a problem via a web front-end and recommends the algorithm and parameters it deems most appropriate, together with an estimate for execution time and accuracy (i.e., the performance to be expected for the relevant metrics). The roles defined in section 4.1 are slightly different from those in PYTHIA II, reflecting PYTHIA II's differing purposes. Instead of *deployers*, PYTHIA II considers 'knowledge engineers': its recommendation mechanism is not directly coupled with any host system, i.e., end-users (experimenter role) have to trigger recommendation manually. The roles of developers and performance analysts are not considered specifically.

For the self-adaptive numerical software (SANS) approach presented in [60], Dongarra and Eijkhout distinguish between users with different levels of expertise: non-expert users benefit from a fully-automated selection, more advanced users may supply additional information to allow for a better selection quality, and expert users may pre-select the algorithm they wish to use. The latter still benefit from using the system, because it will select the most appropriate hardware for executing the desired algorithm [60, p. 4]. The role of developers is not discussed.

In [338], Yu et al. propose an 'Adaptive Algorithm Selection Framework' — not in the sense of a *software* architecture but rather as a description of a generic algorithm selection process for empirical tuning (see sec. 2.7, p. 53). They distinguish between a 'setup' phase and a 'dynamic selection' phase. The setup phase, as the name suggests, is carried out each time the system is installed on a new machine. It executes a factorial experiment (see sec. 3.2.2, p. 68) with an artificial benchmark problem. The selection mapping is then created from the generated data, i.e., this subsumes the steps for data preparation and mining in PYTHIA II, automates them, and restricts them to the implemented methods. The setup phase would be carried out by a *deployer*. The dynamic selection phase occurs at processing time, when an algorithm shall be used — all this is transparent for the end user.

A similarly automated process is implemented for the simulation system TORNADO [42], where algorithm selection is automatically triggered before execution, but prior to this exploitation of knowledge there has to be an exploration phase (i.e., the system adopts a reinforcement learning strategy, see sec. 2.3.2, p. 29). During exploration, users can give manual feedback on the quality of the obtained simulation results.

In [117], Guo puts forward an algorithm selection workflow consisting of seven steps and states that each one *"has some important research problems to solve"* [117, p. 63].

The steps can be easily mapped to those used in PYTHIA II:

1. *Data Preparation*: Guo divides this task into two steps — generating problem instances and collecting performance data.

2. *Data Mining*: this task is split into two steps for preprocessing and two steps for machine learning. Preprocessing first identifies the most relevant problem features (see sec. 2.1.1, p. 14) and then discretizes the data, since the ASP is solved via classification here (see sec. 2.3, p. 24, and sec. 2.6.1, p. 51). Learning consists of a learning step and a meta-learning step (see sec. 2.7, p. 53).

3. *Results Analysis*: this is the final step in Guo's scheme, i.e., PYTHIA II's final step of 'knowledge assimilation' is left out. Guo calls this step 'evaluation': the best selection mapping gets selected and its performance is assessed on test data.

All the above approaches are a little naïve in that they presume the construction of a suitable selection mechanism to be either fully automated or sequential. A notable exception is provided by Smith-Miles in [292, p. 20]. Her conceptual framework distinguishes three phases: the first phase subsumes data preparation and data mining, the second phase relates to knowledge assimilation, and the third phase is concerned with results analysis. Here, however, the results analysis may provide *feedback* to the first phase. If the data mining in phase one yields new insights regarding algorithm properties (e.g., sensitivity for certain problem features), the overall ASP solution process may start over. Phase one (data preparation/mining) and phase three (results analysis) contain activities of the developer and the performance analyst role (see sec. 4.1, p. 79). If, for example, an algorithm is found to exhibit very bad performance on problems where it was not expected, this might hint at a software bug that should be fixed. These fixes then trigger a new reiteration of the ASP solution workflow.

All in all, the discussed solutions have several aspects in common: either they are fully automated, or they distinguish users by expertise (e.g., SANS) or role (e.g., PYTHIA II). All frameworks also distinguish the process of gathering the performance data from the process of analyzing it. In Smith-Miles' framework [292], the analysis is again divided into a manual interpretation of the results — what developers and performance analysts do — and the deployment of a suitable selection mapping, which can be automated. The other approaches regard the ASP solution process as a sequential chain of tasks. While these tasks would be carried out sequentially in an ideal world, it seems more realistic to assume feedback loops after each major step: data preparation may hint at some new features to be considered (and thus, new benchmark problems), data mining and results analysis may identify implementation errors, and even deployment may result in a reiteration, e.g., if the deployed selection mapping turns out to perform worse than expected. This could happen, for instance, if unrealistic benchmark problems and problem features have been used for performance evaluation.

4.4.2. General Architecture

While other systems to solve the algorithm selection problem are a valuable source of inspiration, each of their elements needs to be reconsidered critically: it has to be checked whether it helps to fulfill the technical requirements of the selection mechanism as discussed in section 4.3 (see table 4.1, p. 92).

For example, in [176] it is suggested that the supervisory layer be coupled to the application layer tightly enough to allow the recoding of all interactions between components on the application layer (see fig. 4.5, p. 94, lower right sketch). In case of JAMES II, where components interact to simulate a given model, this would strongly hamper the overall performance: a single simulation run may require *billions* of interactions, e.g., en- or dequeueing an event from a queue. Recording these events just to enable algorithm selection would incur a huge overhead, hence this suggestion violates the scalability requirement. Likewise, the methods related to empirical tuning — the high-level library shown in figure 4.5 (lower left sketch) and the conceptual framework by Yu et al. [338] — both have a strong focus on complete automation and neglect different usage scenarios that are relevant in a simulation

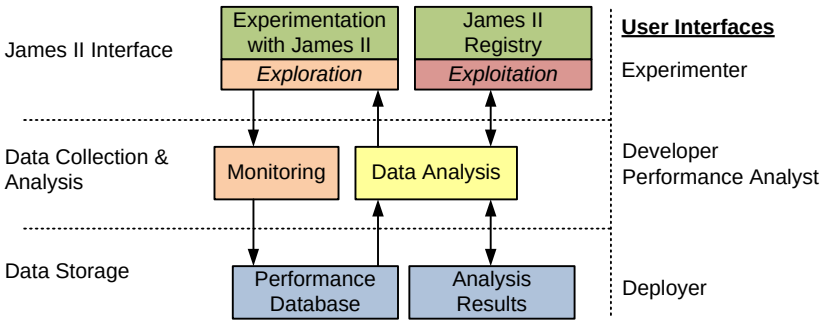

Figure 4.6.: Overall architecture of the simulation algorithm selection framework. Two modules implement the interaction with the host system (Exploration and Exploitation, top layer), one implements data recording (Monitoring, middle layer), one the data analysis, and the two most general modules implement the storage of performance data and analysis results respectively (blue, bottom layer). Colors have the same semantics as in figure 4.5.

context (performance analyst, developer). On the other hand, PYTHIA II is only loosely coupled with its host systems, so that its mechanisms for user interaction cannot be easily adopted for the simulation algorithm selection framework.

Still, there are some relevant commonalities shared by most related approaches, both on a structural and on an interaction level. These should be considered for the development of the framework:

- A separation between the algorithm selection mechanism and the host system (the benefits of which are discussed in [176]).
- A storage for performance data and corresponding meta-information.
- A generic mechanism to add algorithm performance to the storage module, i.e., to support behavioral introspection.
- A module to analyze stored data and draw 'intelligent' conclusions which, are then used for algorithm selection. Selection and data analysis may be realized by separate modules, as in PYTHIA II (upper left sketch, fig. 4.5, p. 94), or by a single entity, as in the SANS layout (upper right sketch, fig. 4.5, p. 94).
- Support for different kinds of users with different goals and responsibilities (in case the tool is not fully automated). The common distinction is between the end-user (experimenter role) and the person solving the ASP with the tool (performance analyst and deployer roles).

Naturally, the aforementioned commonalities are rather general and are not always strictly adhered to. The high-level library (lower left sketch, fig. 4.5, p. 94), for example, maintains performance data only implicitly: it just stores the generated statistical models. Such specifics may or may not comply with the requirements discussed in section 4.3, and will hence be discussed in the following — in this case, dismissing performance data prevents a latter analysis with alternative methods for analysis.

Figure 4.6 shows the overall structure of the simulation algorithm selection framework and its general interfaces with JAMES II. It consists of six essential components, divided into three layers that roughly indicate the degree of interdependence from the host system JAMES II: the topmost layer contains components to *explore* the performance of JAMES II simulators and to *exploit* any knowledge gained by latter analysis. Considering the categorization of algorithm selection methods presented in section 2.6 (see fig. 2.13, p. 52), the exploration modules deals with the data aspect of the ASP, whereas the exploitation module implements the *application* of ASP solutions.

The storages for performance data and analysis results at the bottom layer are rather independent of the host system, although their content is not. Upper and lower layer are connected by an intermediate

layer that is responsible for data collection and analysis. Data collection requires a monitoring module that observes the performance of specific JAMES II components and stores it to the performance database. Finally, the module for data analysis implements methods for actually *solving* the ASP, e.g., those presented in sections 2.3 and 2.5.

Another issue figure 4.6 hints at is the necessity of user interfaces. The user interfaces should be role-specific, to make them simpler and thus easier to use. The level of details on which each interface is likely to focus is indicated by the corresponding role's position in figure 4.6. Note that these user interfaces are not necessarily *graphical*, i.e., some of them may come in the form of an *application programming interface (API)*. Others should be integrated with existing interfaces, e.g., an experimenter should be able to trigger the application of an ASP solution from *within* the general JAMES II user interface. In the following, the discussion on suitable user interfaces — as important as they are — will be restricted to *program* interfaces. The next paragraphs detail the general responsibilities for each part of the framework. Their specific structure and implementation are described in the following chapters.

Performance Data Storage & Monitoring

The performance data storage manages all data to be considered for solving the simulation algorithm selection problem. The theoretical framework of Rice [272], as discussed in section 2.1 (p. 13), already defines the most important entities; these need to be mapped onto more concrete concepts from modeling and simulation. Additionally, it is important to discuss different ways of defining and obtaining these entities for JAMES II — the goal is, again, to automate as much of the process as possible. Afterwards, the recorded data is added to the database. As the arrows in figure 4.6 suggest, the monitoring device will be specific to the host system JAMES II, whereas the performance data storage will be more compatible with other systems. Chapter 5 details both modules.

Data Analysis

The main responsibility of the data analysis module is to generate selection mappings from the empirical data provided by the performance data storage. In other words, it has to *solve* the simulation algorithm selection problem (see def. 2.1.2, p. 15). Several research communities focus on the analysis of empirical data from different points of view, most importantly statistics and machine learning (see sec. 2.3, p. 24). Consequently, there are many potentially suitable methods and tools available to solve the ASP — which makes the *integration* of them the most important requirement of the data analysis module. All integrated methods should be accessible via a unified interface that is tailored to the creation of selection mappings. Besides the generation of selection mappings, these also need to be evaluated and compared by different performance metrics, e.g., how often a mapping selects the best algorithm. All these features need to be realized by the data analysis module. Its main components are described in chapter 6.

Extended Experimentation Layer

The JAMES II experimentation layer as such does not rely on algorithm selection. However, solving the ASP requires considerable amounts of empirical data that should be obtained via automated experimentation. The experimentation layer should therefore support the large-scale exploration of algorithm performance on various models. As the enhancements to the experimentation layer cannot be explained without understanding its basic structure, chapter 7 will first give some details on its general workings, before describing the extensions that were specifically added to better support the collection of empirical performance data. One technique presumes a specific property of benchmark models; their construction is thus covered as well. As it turns out, it is already on this level that simple solutions for the ASP can be realized.

Extended Registry

Finally, chapter 8 discusses how the JAMES II registry can be extended to make use of the selection mappings generated by the data analysis module. This is in some sense the most crucial element, as it allows a JAMES II user to apply the ASP solutions that are available. However, it is also the most system- and role-specific element — both other JAMES II-specific modules, monitoring and extended support for performance experiments, are also used by developers and performance analysts to carry out their tasks. An extended registry, on the other hand, needs to be configured by a deployer. Interestingly, the knowledge on plug-in performance — implicitly contained in the selection mappings — is not the only kind of meta-data that the registry may rely on. By making the life-cycle of a plug-in explicit and generalizing storage and usage of meta-data in the registry, the *robustness* of flexible simulation systems can be enhanced as well (see sec. 2.4.3, p. 36). This resolves the vulnerability of the current selection process with respect to malicious plug-ins, as discussed in section 4.2.3 (p. 87). The registry's management modules for meta-data should be regarded as a part of the data analysis module (see fig. 4.6, p. 97), as they also manage the deployed selection mappings, i.e., they interface the storage of analysis results.

4.5. Summary

This chapter first discussed some relevant use cases for simulation algorithm selection (sec. 4.1, p. 79). Then, the modeling and simulation framework JAMES II and its most important characteristics regarding algorithm selection have been introduced (sec. 4.2, p. 81). Section 4.3 (p. 89) described the technical requirements a simulation algorithm selection mechanism for JAMES II has to fulfill. Finally, some systems with related goals have been surveyed briefly (sec. 4.4.1, p. 92), to get an idea of their overall design. The major commonalities among these approaches served as a starting point for a blueprint of the simulation algorithm selection framework (SASF). Its structure was motivated and outlined in section 4.4.2.

Requirement	**Relevant for**	**PDS**	**Analysis**	**E-Exp**	**E-Reg**
Separation of Concerns	Deployer	+	+	+	++
Scalability w.r.t. #Algorithms	PA / Dev	++	+	+	++
(Automated) Behavioral Introspection	PA / Dev	+	+	++	∘
Scalability w.r.t. #Performance Data	PA	++	++	∘	∘
Behavioral Intercession	Experimenter	∘	++	+	++
Meta-Learning / Performance Evaluation	Deployer	∘	++	∘	+

Table 4.2.: This table indicates the most important relations between the identified requirements, user roles, and modules.

To give a brief visual summary of the most important aspects discussed in this chapter, table 4.2 associates the requirements from table 4.1 with user roles and SASF modules. Most requirements are particularly important for performance analysts (PA) and developers (Dev), because large-scale data collection is a prerequisite for the generation of good selection mappings (as already discussed in ch. 2, p. 13). Deployers need tools to assess the effectiveness of the generated mappings and also benefit the most from a clear separation of concerns, as this allows them to easily replace parts of the overall system (or to deactivate them altogether, in case of malfunctioning). The ultimate goal of the system, however, is to provide *experimenters* with a simulation system that supports automatic behavioral intercession. The columns on the right of table 4.2 indicate the importance of the requirements for the basic modules introduced in section 4.4.2: performance data storage & monitoring (PDS), data analysis & analysis results storage ($Analysis$), extended experimentation layer ($E\text{-}Exp$), and extended registry ($E\text{-}Reg$). The degree of importance varies between low (∘) and high (++). Scalability with

respect to algorithms and the separation of concerns are clearly *cross-cutting* concerns, i.e., they need to be considered in the development of *all* sub-systems. Others, such as behavioral intercession, are only relevant for some of them. The next chapters will cover each sub-system in more detail, and discuss how these requirements can be fulfilled.

5. Storage of Performance Data

> *This step is simple to explain, but nontrivial to actually perform. In the case studies that we have performed, we have found the collection of data to be very time-consuming both for our computer cluster and for ourselves.*
>
> Leyton-Brown et al. [205, p. 9]

This chapter describes how the ASP entities discussed in section 2.1.4 (p. 22) can be recorded and stored, i.e., it covers the parts of the simulation algorithm selection framework that are highlighted in figure 5.1. Storing performance data in some common, readily accessible format is one of the major premises for reproducible and comparable performance results. Gent et al., for example, advise researchers to store all their published data for later re-evaluation [101]. The advantages of this approach become apparent in the work of Gagliolo and Schmidhuber [96], who reuse publicly available performance data published by Leyton-Brown et. al. [204] to test a new method for algorithm selection. In a similar vein, Adve et al. state that their performance modeling and prediction tool POEMS (see sec. 3.3.2, p. 73) *"[...] could not be built without a database as a searchable repository for a wide spectrum of model and performance data"* [3, p. 1035].

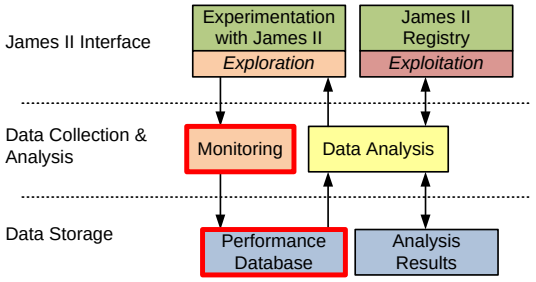

Figure 5.1.: SASF overview (see fig. 4.6, p. 97), red borders denote elements discussed in this chapter.

Consequently, there are some general attempts to provide storage solutions for performance data on software (or pieces thereof), particularly in the field of performance analysis tools briefly surveyed in section 3.3.2 (p. 73). In [19], Berry et al. present a *performance database server (PDS)* that allows to manage benchmark scores for various machines and to associate them with corresponding publications. The motivation for PDS, however, stems more from comparing different kinds of machines and not from comparing different kinds of algorithms on the same machine. For each type of benchmark, there is a dedicated database table whose structure reflects the scoring system of the benchmark. While this makes the PDS flexible enough to store any kind of benchmark data and does not hinder manual interpretation, it makes an automated result analysis — as required for algorithm selection — quite hard.

A more algorithm-centric viewpoint is, for example, taken by the performance data management framework PerfDMF [151]. It stores detailed performance data that can be analyzed with PerfExplorer [150] (see sec. 3.3.2, p. 73). The Prophesy [301] tool also features a performance database and allows to record performance data in a detailed manner, i.e., up to the level of functions and code segments. Then again, this level of detail is not required for the performance evaluation of simulation *algorithms*, which typically consist of many functions and code segments.

Storing large amounts of additional data without any surplus could hamper the applicability of the system, even for performance experiments of a moderate size. This violates the requirement of scaling well with the amount of available performance data (see tab. 4.2, p. 99). Maximal scalability is only achieved if there is an *efficient* way to store *just* the data that is necessary to solve the algorithm selection problem. The collection of relevant performance data from sound experiments is a prerequisite task (see sec. 3.1, p. 61). A suitable performance storage should allow to reproduce former performance experiments and to retrieve the recorded algorithm performances easily. To combine these

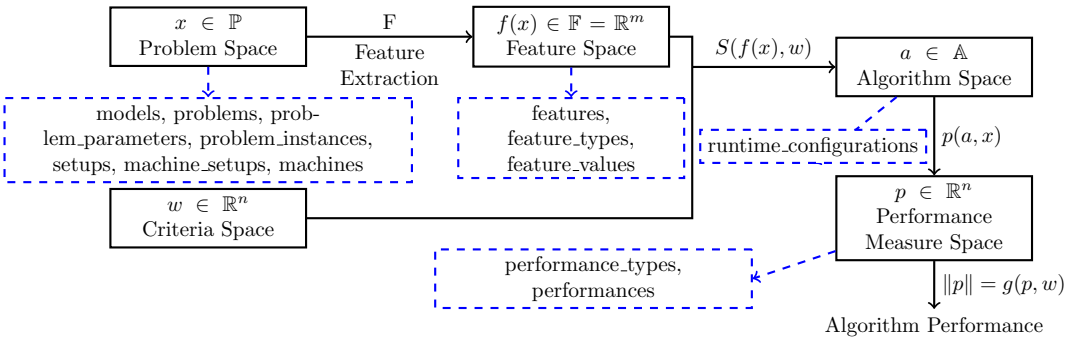

Figure 5.2.: The algorithm selection problem as defined by Rice [272], and its mapping to concrete database entities (dashed, blue). The criteria space needs no representation within the database, as these preferences need to be given by the user (cf. fig. 2.3, p. 22).

features with low-level data on sub-routines is possible in principle, but not particularly necessary from a practical point of view. Developers and performance analysts should use the SASF to find out *which* algorithm works best on *which* problems — for finding out *why* this is the case, they may still resort to profiling tools or systems like PerfDMF.

Hence, existing systems for algorithm selection — e.g., PYTHIA II — rely on custom databases that are tailored to their needs and only store the entities they work with. Similarly, Dongarra and Eijkhout propose a 'history database' for their concept of a self-adaptive numerical software (see sec. 4.4.1, p. 92) and state that *"[c]ategorization of performance and problem 'metadata' into relational databases should be based on the application domain [...]"* [60, p. 128]. It therefore seems advisable to implement a dedicated performance data storage, targeted at simulation algorithm selection. This also allows to offer extended support for a rather simulation-specific experimentation challenge discussed in chapter 3, namely observing the performance of algorithms for stochastic simulation (see sec. 3.2.1, p. 66). An earlier version of the work described in this chapter has been presented in [70].

5.1. The SASF Performance Database

The overall database structure can be derived from the theoretical formulation of the algorithm selection problem (sec. 2.1, p. 13) and its mapping to the simulation domain (fig. 2.3, p. 22). The database entities that will be detailed in the following are depicted in figure 5.2.

Most effort is needed to define the problem space $\mathbb{P}$ in a coherent and general manner. This is because the problem definition does not only subsume the input of a simulation algorithm (e.g., models), but also the available infrastructure that shall be used for its execution (e.g., setups, machines). The storage schemes for performance and feature space are rather similar. Both are designed to be extensible in the future, i.e., they allow the definition of arbitrarily many *types* of features and performance measures (i.e., additional dimensions of feature space and performance space). Performance measurements are associated with the application of a specific simulation algorithm to a specific simulation problem instance, i.e., the execution of a single simulation run. The run is represented by an additional *application* entity, which does not correspond to the formal entities shown in figure 5.2 and is hence described separately in section 5.1.1 (p. 106). The algorithm space is represented by simply enumerating all kinds of available plug-in combinations, which are called 'runtime configurations'. Section 5.1.1 details the definition of the database entities. Section 5.1.2 briefly discusses the generality of this layout, i.e., to which degree it can be used by other simulation systems.

The challenge of efficiently storing large amounts of performance data should not be underestimated. Besides identifying and implementing suitable entities, the data sink should be easy to handle, as independent of external software as possible, and also efficient. The actual implementation is covered in section 5.1.3.

5.1.1. Entities

Problem Space ($\mathbb{P}$)

The problem space $\mathbb{P}$ of the algorithm selection problem (see fig. 5.2) simply contains all simulation problems that can be solved by the available simulation algorithms in $\mathbb{A}$. It consists of two large subspaces: one is the space of all potential input models, the other is the space of all potential hardware setups that might be available, i.e., the sets of machines to be used and the network equipment by which they communicate.

Model Description Storing models directly into a database would mean to create (and curate) a model database instead of a performance database, which is a challenging research topic on its own. Developing a proper model database has to fulfill many additional requirements (e.g., with respect to searchability).

To circumvent these difficulties and also to allow referencing models from various (external) sources, a model is merely defined by a *uniform (or universal) resource identifier (URI)*, a common standard for specifying the location of a resource on the Internet. This does not only comply to the input that is expected by the experimentation layer of JAMES II [132], but also ensures that a model can be stored on arbitrary resources — including the file system and model databases. As long as the corresponding URI can be resolved, the model is accessible. Additionally, a *model* entity in the performance database has a name, a description, and a certain type. Different types of benchmark models are discussed in section 7.3.1 (p. 145); the categorization is only used for documentation so far.

Most models have several parameters that can be adjusted by the experimenter. A model defining chemical reactions, for example, may allow to specify the volume, the temperature, and the pressure. Furthermore, models are usually simulated for a predetermined time span (given in simulation time, see fig. 3.2, p. 69) or until a termination condition is true. For example, a reaction network might be simulated until it arrives at a chemical equilibrium. Whatever this rule for stopping the simulation is, it clearly belongs to the overall problem description, along with the model parameters. These informations are bundled together to a *problem* entity. It covers all aspects of a single simulation run that an experimenter typically has to define: which model is simulated, with which parameters, and for what time span (or until which condition holds true). Note that a simulation problem, i.e., an element of the problem space $\mathbb{P}$, also contains a description of the hardware setup that shall be used.

Infrastructure Many publications have already shown the considerable impact of hardware architecture and operating system on algorithm performance, particularly when it comes to execution speed. In [188], Lagoudakis and Littman observe that the optimal decision strategies for sorting algorithms — based on the input size only — differ between Solaris (on a SPARC platform) and Linux (on a Pentium platform). LaMarca and Ladner show in [191] that caching hierarchies have a strong impact on the performance of sorting algorithms, and that the reduction of (time-consuming) cache misses leads to theoretically inferior algorithms that *outperform* theoretically superior ones by a large margin. For example, multi-mergesort, a memory-optimized version of the original algorithm, was up to 70% faster than the original mergesort, despite executing 75% more instructions. Similar effects can be observed for other memory access mechanisms as well, e.g., paging [175]. In [318], Vuduc et al. investigate the performance of algorithms for matrix multiplication. They find that cache sizes and the number of CPU registers correlate heavily with the optimal matrix size for each multiplication algorithm. Apart from the impact that hardware has even on rather simple algorithms, new simulation approaches may also exploit specialized hardware that is not always available [253, 280, 287].

All these points suggest that there is no easy way out; hardware specifics do matter and have to be considered as an independent variable when it comes to analyzing algorithm performance. Furthermore, McGeoch concludes that *"[t]here is no known general method for making accurate predictions of performance in one programming environment based on observations of running times in another"* [228, p. 305] — the reasons for this have already been detailed in section 3.1.1 (p. 62). Hence, many

practical algorithm selection approaches either rely entirely on performance tests executed on the target platform, or they introduce an additional calibration at installation time (see [25, 338] as well as the discussions in sec. 2.6.1, p. 48, and sec. 4.4.1, p. 95). Whether or not data from other machines or platforms is considered depends on the nature of the measurements and should *not* be restricted by the performance database (e.g., see tab. 5.2, p. 109).

None of the surveyed algorithm selection approaches associates formal hardware definitions with performance data in order to transfer the results automatically to the target platform. The definition of infrastructure will therefore follow a very simple pattern that consists of only two entities: a *machine* has a name, a MAC address (for easier identification in a network), a description, and a benchmark score. Currently, only a single benchmark score (for Java SciMark [263]) is stored; of course this could be easily extended to whole lists of benchmark scores, similar to the PDS system discussed before. Machines are aggregated to *setups*. Besides a number of machines, a setup contains information on the network infrastructure, given as textual information on topology or equipment (e.g., *"hyper-cube"* or *"Netgear switch"*). Furthermore, a rough indicator of the network speed can be given as an integer value. For example, this could denote the average round-trip time of a packet in milliseconds. Such figures, of course, give nothing but a rough idea about the network connection quality. Hence, the data stored on the network setup and the available machines is intended for manual interpretation, not for any kind of automated performance analysis.

Problem Instance Some performance metrics — like execution time — vary from execution to execution (see sec. 2.3.1, p. 25, and sec. 3.2.1, p. 66). This is due to the unobserved state of the hardware when execution started, and also due to stochastic models or simulation algorithms. To deal more efficiently with the latter source of stochastic noise within the recorded performance data, an intermediate entity named *problem instance* is introduced.

Problem instances are very simple entities: they are defined upon a certain problem entity and only contain additional data to specify the sequence of random numbers that is generated by the host system, i.e., which RNG is used and with which seed it is initialized (see sec. 3.2.1, p. 66). In JAMES II this is controlled by defining the corresponding RNG factory and the parameter block with which it is initialized [74]. This allows to use common random numbers, i.e., the simple yet commonly used variance reduction technique already discussed in section 3.2.1 (p. 66). If two runtime configurations are to be compared, for example, initializing the host system to use common random numbers will make the comparison more precise. However, this technique is not always useful — e.g., in case of deterministic approaches — in which case there will only be a single problem instance per problem entity.

Algorithm Space ($\mathbb{A}$)

Algorithms are not necessarily monolithic entities: in JAMES II, a simulation algorithm may rely on numerous auxiliary algorithms and data structures to fulfill the task at hand. Hence, it does not suffice to store the name of an algorithm or the name of its class file — there is additional structural information that must not be neglected. This is most relevant for performance analysts and developers who want to examine the impact of a certain sub-algorithm on overall performance. How can the structure of such a *combinatorial* perspective on algorithms[1] be made explicit, so that it can be stored? In the SASF, this is done by introducing the notion of *selection trees*. They define the hierarchical relationship between the algorithms involved in a simulation run (see fig. 4.3, p. 85), and also with which parameters these algorithms have been instantiated. This makes different setups distinguishable for later analysis and also ensures the reproducibility of the experiments:

Definition 5.1.1 (Selection Tree)**.** *A selection tree is a directed tree* $s = (V_a \cup \{r\}, E)$ *with:*

- $V_a \subseteq \mathbb{N} \times A \times \mathcal{P}$ *is a vertex set, excluding the root* r*. The numbers from* $\mathbb{N}$ *specify an order of the sub-algorithms from* A*, i.e.,* A *denotes the set of algorithms eligible for combination (e.g.,*

[1] See categorization of ASP approaches w.r.t. their consideration of algorithms in sec. 2.6.1, p. 49.

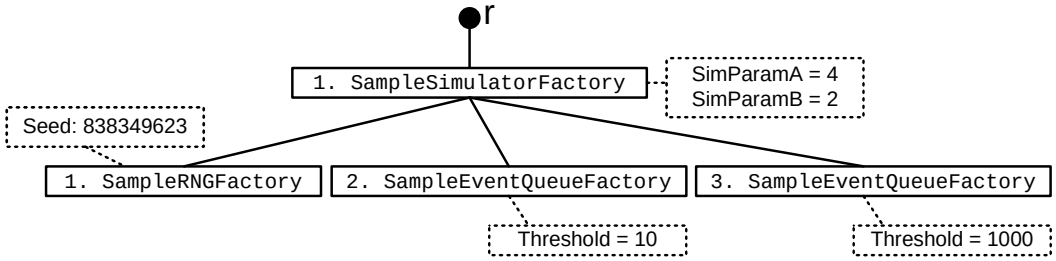

Figure 5.3.: A sample selection tree. Parameter maps are denoted by dashed boxes, algorithms and their numbering is denoted by solid boxes. The numbering is necessary to provide unambiguous parameterization if two components of the same type are used by the same component. In the above example, the `SampleEventQueueFactory` is used twice to create event queues with different threshold parameters, i.e., for different tasks.

JAMES II *plug-ins).* $\mathcal{P}$ *is the set of all* parameter maps $\tilde{p} : String \to \mathbb{R}$*, which define the parameters to initialize the algorithm with.*

- $E \subset V_a \cup \{r\} \times V_a$ *is the set of edges.*

Furthermore let $sub_a(v) = \{v' | (v, v') \in E\}$ *with* $v \in V_a \cup \{r\}$ *be a set of sub-algorithm vertices, i.e., the child nodes of vertex* v*. The numbering among the child nodes must not have any gaps:*

$$\forall v \in V_a \cup \{r\} : (\forall v' = (n', a', p') \in sub_a(v) : n' = 1 \vee (\exists v'' = (n'', a'', p'') \in sub_a(v) : n'' = n' - 1))$$

In other words, a child vertex numbering is either 1 *or there is a sibling numbered with a direct predecessor* $n - 1$*. Furthermore, numbers must be unique among siblings:*

$$\forall v \in V_a \cup \{r\} : (\forall v' = (n', a', p') \in sub_a(v) : (\nexists v'' = (n'', a'', p'') \in sub_a(v) - \{v'\} : n'' = n'))$$

Selection trees can now be used to give a concrete definition of the algorithm sub-space $\mathbb{A}_x \subseteq \mathbb{A}$ that is applicable to a given simulation problem $x \in \mathbb{P}$, provided that there is a predicate

$$Applicable(s, x) \iff \text{selection tree } s \text{ is applicable to problem } x \in \mathbb{P}$$

so that

$$\mathbb{A}_x = \{s | s = (V_a \cup \{r\}, E) \wedge Applicable(s, x)\} \tag{5.1}$$

Usually, such a predicate is implicitly defined by the host system, e.g., in case of JAMES II it is implemented by the factory filtering mechanism (see sec. 4.2.1, p. 85). The union of selection tree sets for all simulation problems in $\mathbb{P}$ thus represents the algorithm space $\mathbb{A}$ for JAMES II (cf. fig. 4.4, p. 88). The name 'selection tree' stems from the fact that each vertex (except the root r) represents an algorithm (i.e., plug-in) that has been *selected* from the set of combinable algorithms (A). A sample tree is given in figure 5.3.

Selection trees are the only central database entities that are based upon other JAMES II entities, namely the abstract factory and the concrete factory corresponding to a plug-in and its type. However, as selection trees are simply stored in binary format, i.e., the properties of the object are irrelevant to the database itself, they can be easily adjusted to store configuration information for another host system. A selection tree just defines the *specific* configuration of the host system that was used to solve a simulation problem. Making this configuration explicit and mapping it onto the formal structure given in definition 5.1.1 should be unproblematic for any other simulation system. In case algorithms

are merely regarded as monolithic or monolithic-parameterizable, their selection trees will just consist of the root and a single additional vertex, i.e., $|V_a| = 1$. While the above notion of a selection tree is unique to the SASF, others kinds of tree structures have already been used to illustrate a nested selection of algorithms, e.g., by Lagoudakis and Littman [188].

Runtime Configuration Selection trees are not managed as such; they are part of a *runtime configuration* entity and stored in a single field that may hold any *binary large object (BLOB)*. Besides a selection tree, a runtime configuration contains a version field that is automatically incremented when a developer re-issues a new version of an algorithm for automated performance re-evaluation (a feature introduced in section 7.3.4). For the same reason, each runtime configuration also contains a flag that determines if this particular entity relates to the most current version of the selection tree. The rationale for this is that algorithm implementations may change so much — e.g., due to optimization or bug fixing — that all past performance results become outdated. To keep both old and new performance data in the same database, distinguished by the different runtime configurations they are associated with, may have some advantages (see sec. 7.3.4, p. 156). Therefore, multiple runtime configurations with different version numbers but the same selection tree may exist. Only one of them is flagged to be the current version. To better keep track of the version history, each runtime configuration also contains the date at which it was added to the data storage.

Applications (−)

Although not being part of any fundamental ASP entity (fig. 5.2, p. 102), it is important to explicitly store the execution of single simulation runs. A simulation run is viewed as the *application* of a runtime configuration to solve a simulation problem instance. Applications are in some sense the most central entities of the performance database, as each one represents an *association* between problem space $\mathbb{P}$ and algorithm space $\mathbb{A}$. An application refers to a problem instance entity and a hardware setup entity to specify the element from $\mathbb{P}$, and also to a runtime configuration entity to specify the element from $\mathbb{A}$. Another field stores the date and time at which the application was inserted into the database, i.e., at which time the execution finished. This helps to manage large amounts of performance data: for example, if performance experiments within a certain time period were executed on a busy machine and are hence considered biased, these should be easy to query (and delete) from the database. Finally, application entities need to link the execution they represent with the data that it generates, i.e., observed simulation outcomes. While observing simulation data (and the performance impediment of doing so) are often neglected in simulation algorithm comparisons, some performance metrics are *defined* on the simulation output. For example, measuring the accuracy of an approximative simulation algorithm may require to compare its results to an exact solution. When it comes to simulation results data bases, basically the same arguments as with model databases apply: these tools have their own requirements and challenges, which should not be emulated by a storage system that serves a different purpose. In fact, it is even possible to apply the same kind of solution in this context: instead of directly storing the data observed from a simulation run, only a *pointer* to the data is stored. This is implemented by defining an additional binary field that may contain a Java object implementing the `ISimDataProvider` interface defined by the SASF. While this interface relies on other interfaces defined by JAMES II, the performance data storage makes *no* assumptions about the implementation of the *simulation data provider*. The default implementation allows to retrieve simulation output from any JAMES II data storage (see [132]).

Feature Space ($\mathbb{F}$)

A proper definition of simulation problems in the data storage is important for later reproducibility, yet algorithm selection as such depends on the problem *features* that are observed [272, 292]. Although one may intuitively link features to model entities, these need to be associated with *application* entities: features may also describe certain aspects of the infrastructure that is used, or structures that are

randomly created at model initialization and hence require knowledge of the seed and the RNG implementation that are used.[2]

In contrast to the ASP entities described before — problems ($\mathbb{P}$), algorithms ($\mathbb{A}$), and also specific simulation run executions — the structure of the feature space $\mathbb{F}$ is highly domain-dependent (e.g., see survey in [292]) and hence cannot be predefined. In [205], Leyton-Brown et al. point out two generally important criteria for a feature:

- It should be computable for every problem instance (i.e., no reliance on prior information).
- It should be easy to compute, i.e., much faster than the pay-off for selecting a good algorithm.

Both criteria simply ensure that the given features can be actually used in a practical setting: if a feature cannot be easily computed for all simulation problems of interest, algorithm selection may not yield any performance gain. For example, Dongarra and Eijkhout note that for matrix multiplication, both the symmetry of a matrix as well as its positive definitiveness might be relevant features for algorithm selection [60] — however, while the symmetry of a matrix is easy to test, its positive definitiveness is not. Table 5.1 summarizes this and other features from various problem domains, including parallel and distributed discrete-event simulation (PDES). Features that can be considered for chemical reaction networks are discussed in chapter 9.

Domain	**Features**	**Described in, e.g.,**
Matrix Multiplication	Matrix symmetry, positive definiteness	[60]
Meta-Learning	Attribute and class entropy, landmarking (see sec. 2.7, p. 55)	[257]
Sorting	Entropy and standard deviation of keys	[206]
Combinatorial Search	Graph properties, solution of relaxed problem	[205]
Network Communication	Message size, call type, number of processors	[311]
Partial Differential Equations	Kind of equation, domain, boundary equations	[146]
PDES	LP connections and communication rate, time-stamp increment, message delay in network	[45]

Table 5.1.: This table gives some examples of domain-specific features that have been proposed or considered for algorithm selection. The given references are not complete; others have proposed similar features for most of the above domains. For example, very similar features of partial differential equations have also been discussed in [270] — the given references are hence to be understood as pointers to exemplary studies. Furthermore, most of the features named above refer to whole *groups* of more concrete — i.e., quantitative — features. For example, Leyton-Brown et al. consider more than 30 distinct problem features in [205].

Many of the above domains — e.g., sorting, matrix multiplication, and network communication — have a certain overlap with simulation algorithms: a simulation algorithm may have to multiply matrices, sort events, and propagate those over the network. It is unclear how many (and which) of the above features need to be included for which kind of simulation-related algorithm selection problem. Moreover, specific models like chemical reaction networks may give rise to entirely new features, and it is usually unclear which of them are relevant for algorithm selection (see def. 2.1.3, p. 15). It is therefore important to store all features in a generic way, and also to allow a later addition of new features. This is done by defining feature types, features, and feature values as distinct database entities.

[2] As mentioned in section 2.4.1 (p. 34), it is generally assumed that all instances of a simulation problem have the same features, i.e., stochasticity during model initialization does not play a role. However, the structure of the database is able to cope with instance-specific problem features as well.

Feature Type A *feature type* defines a specific dimension of the feature space $\mathbb{F}$ and consists of a name, a verbal description , and additional data on how the feature can be calculated for a given simulation problem. The latter is done by internally relying on a free-form string. The interface of the performance data storage, however, assures that this field is used for the fully qualified class name[3] of a class that is able to extract the feature. The process of extracting features from a simulation problem is handled by a new *feature extraction* plug-in type, with `FeatureExtractorFactory` as a base factory. Concrete feature extraction factories create implementations of `IFeatureExtractor`, an interface that receives a `ParameterBlock` containing all simulation problem specifics and then returns a set of (*feature name*, *value*) tuples, where the feature name is unique. This allows to define a multi-dimensional feature space by implementing a single feature extractor. Furthermore, features are regarded as binary objects and may hence contain structured data. This allows to store features that belong together within a single feature object, which goes well along with object-oriented programming. How and when feature extraction is executed for a given simulation problem is described in section 5.1.3.

Feature & Feature Value A *feature* entity results from applying a feature extractor to a simulation problem. It merely subsumes all *feature values*, i.e., name-value pairs, that have been extracted by from a simulation problem. As with simulation data providers or model parameters, all feature values are stored as binary objects and may hence contain data in arbitrary format.

Performance Space ($\mathbb{R}^n$)

At first sight, storing performance data seems almost trivial when compared to the other entities of the ASP, particularly algorithms and features. After all, most simulation developers are focused on improving the execution time, which can be easily stored as a floating point number. However, mere execution speed may not be the most important aspect for users, and neither the most significant one when it comes to *analyzing* algorithm performance (i.e., for the developer and performance analyst role). As Moret puts it: *"If there is one universal piece of advice in this area, it is always look beyond the obvious measures!"* [235, p. 11]

Execution time itself belongs to a whole *family* of performance metrics that are all concerned with the consumption of computational resources (see sec. 2.1.4, p. 22): memory, network bandwidth, CPU cycles, GPU cycles, or accesses to a disk storage (to name a few). Such measures are also referred to as *consumption performance measures*. From this more detailed perspective, the overall execution time is nothing but an aggregated measure of all operations on the critical path (see sec. 3.3.1, p. 71). For example, it neglects how many additional work could be done on the same machine in parallel, or the overall amount of electrical energy that is consumed. While alternative metrics to characterize resource consumption have been proposed, e.g., Knuth suggested to count memory references (*mems*, see [235]), none of them is as widely used as measuring the overall run time.

Besides resource consumption — which should be minimized — there are several other aspects of an algorithm implementation that are of practical relevance. Not all of them are normally considered as its performance:

- *(Numerical) Stability*: Some algorithms only work for certain sub-classes of input and otherwise produce wrong output. In simulation, this is the case for numerical integration schemes, which can be broadly divided into implicit and explicit schemes, all having distinct regions of numerical stability (see [34, p. 34 et sqq.]).
- *Accuracy*: An algorithms' accuracy is of interest in case it only approximates the true solution. In simulation, this again applies to numerical integration algorithms (e.g., [272, p. 82]), but also to some simulation algorithms for chemical reaction networks (see sec. 1.3.1, p. 4). In contrast to stability, which determines to which problems an algorithm is applicable, its accuracy measures how close its calculations approach the true solution.

[3] Put simply, a fully qualified class name combines package name and class name, e.g., `my.package.MyClass`.

- *Robustness*: It is sometimes important to not only consider average algorithm performance, but also the extent with which the performance *deviates* from problem to problem. Large deviations may make an algorithm less useful in circumstances where user interaction or some other time constraints are relevant.

- *Non-Quantifiable Aspects*: Weihe claims that *"[...] non-quantifiable characteristics are often more important [...]"* [325, p. 5] than quantifiable ones. As examples he names flexibility (i.e., how easily an algorithm can be adapted to solve other problems), error diagnostics (i.e., which information the algorithm provides in case it cannot find a solution), and user interaction (i.e., how it copes with additional constraints and data available at runtime). The latter point also touches upon the robustness issue, i.e., it may be important in practice to guarantee a certain solution quality within a certain time span. Similarly, Houstis et al. highlight the importance of an algorithm's reliability, its portability, and its documentation [146].

In the above list, the aspects are ordered by their importance to an algorithm's applicability. If it is unknown in which region an algorithm is numerically stable, it is of very limited use as all results could be completely wrong. If it is unknown how accurate the algorithm is, results may not be extremely wrong but still deviate strongly from the true result. Yet, both aspects are only of interest for certain problem domains. The other issues — robustness, flexibility, error diagnostics, interaction, reliability, portability, and documentation — are very difficult to quantify and require a careful problem-dependent reconsideration.

Domain	**Performance Metric**	**Described in, e.g.,**
Sorting	CPU cycles per key	[206]
Compilers	Code size	[319]
Artificial Intelligence	Number of backtracks	[114]
Random Number Generators	Randomness (super-uniformity, LIL-uniformity)	[125]
Reinforcement Learning	Regret	[11]
Parallel Algorithm Portfolios	Overhead	[97]
Hardware	Area, power consumption, critical path	[262]
Simulation Experiments	Statistical efficiency, user utility	[124, 239]
PDES	Scalability, efficiency, speed-up, processor utilization, rollback frequency, rollback length	[45, 124, 238]

Table 5.2.: This table gives some examples for performance measures that have been considered in algorithm selection or performance analysis. As in table 5.1, the given references are to be understood as pointers to exemplary studies.

Apart from these rather general facets of performance, there are also many domain-specific performance measures. Table 5.2 lists some of them. Some measures are concerned with solution quality, i.e., features of the generated output (size of generated code , randomness), while others are concerned with the usefulness from a user perspective (utility), the amount of *unnecessary* resource consumption (e.g., rollback frequency, statistical efficiency), or merely reflect overall execution time (CPU cycles per key).

One can also observe that the meaning of a performance facet is not always coherent across literature. This may occur when concepts are inherently hard to describe (even mathematically), as for the randomness of RNGs [125]. In other cases, e.g., in case of 'efficiency', the ambiguity just stems from different perspectives: for example, Cortellessa and Quaglia [45] define the efficiency as $E = 1 - F_r \cdot L_r$, with F_r being the roll-back frequency and L_r being the average roll-back length (see sec. 1.3.2, p. 6). Thus E denotes the (averaged) share of committed (i.e., non-rolled-back) events in relation to all simulated events. In [124], however, Heidelberger defines efficiency as $e_p = \frac{\alpha_p}{p}$ with $\alpha_p = \frac{t_1}{t_p}$

being the speed-up of using p processors (execution time t_p) instead of just one (execution time t_1). Although both notions of efficiency seem intuitive and are defined for a parallel and distributed discrete-event simulation, they strongly differ in terms of what they take into account and what they measure. For example, the efficiency of a conservatively synchronized PDES execution would always be 1 following Cortellessa and Quaglia (which use this measure to assess *optimistic* synchronization schemes), but not necessarily with Heidelberger's. Similarly ambiguous notions of PDES performance are scalability, speed-up, or processor utilization. It is therefore mandatory to not only *name* the type of a performance measurement stored in the database, but also to accurately define *how* it was measured.

Moreover, one should note that adding new performance measures does not always add more information: e.g., if Heidelberger's notions of efficiency and speed-up are adopted, it is unnecessary to store *both* speed-up and efficiency, given that the number of used processors is already defined by the chosen setup: both measurements are proportional to each other with a constant factor $\frac{1}{p}$, i.e., they are linearly dependent. In [138], Hockney even claims that speed-up, the performance per processor in floating point operations per second, or the efficiency are *all* misleading as performance metrics — what would ultimately count is the number of problems solved per second. Speed-up is only a relative measurement between the performances of sequential and parallel execution and thus, he argues, not always transferable to other architectures. Similarly, performance per processor and efficiency may hide runtime increases due to too much parallelization — the potential bias of all measures is exemplified by a particle simulation that serves as a supercomputer benchmark.

Performance Type & Performance All these issues — lack of common definitions, problem-dependent metrics, difficulties in measuring and results interpretation — make it necessary to make performance measuring as flexible as feature extraction, and also as reproducible. This is supported by allowing the definition of arbitrary *performance types* and associating each observed *performance* with one. To do so, a new *performance measurer* plug-in type is introduced, which provides components that implement the `IPerformanceMeasurer` interface. A text field is used to simply store the fully-qualified class name of the concrete measurer factory. As with feature types, each performance type also has a name and a description.

The `IPerformanceMeasurer` interface itself is defined upon `SimulationRuntimeInformation` instances and returns a floating point number. This is in accordance to Rice's definition of a performance space as $\mathbb{R}^n$. `SimulationRuntimeInformation` is a JAMES II-specific class that contains a description of the simulation run's setup, its unique ID, and additional measurements like the execution times required for model initialization and simulation. Section 5.2 describes when and how performance measuring is triggered.

Finally, it should be noted that the above components allow to associate algorithm performance with the *application* of a runtime configuration to solve a *single* simulation problem instance. Under specific circumstances, however, it is more reasonable to associate performance with *problem* or *problem instance* entities.[4] An association with problem instance entities is sufficient in case the performance metric is hardware-independent, e.g., the accuracy of a deterministic yet approximative simulation algorithm (like a numerical integrator) should be stored on that level. An association with problems instead of problem instances is sufficient in case the performance metric is hardware-independent *and* considers an algorithm's performance over multiple multiple applications. A concrete example for such a performance measure is the accuracy of SSAs (see sec. 1.3.1, p. 4, and discussion in ch. 9, p. 177). While these more abstract kinds of performance types are not supported by default, an extension of the scheme in a similar manner is straightforward.

[4] The same reasoning also applies to features. Some features may be the same for all applications of a problem instance, or for all problem instances associated with a problem entity.

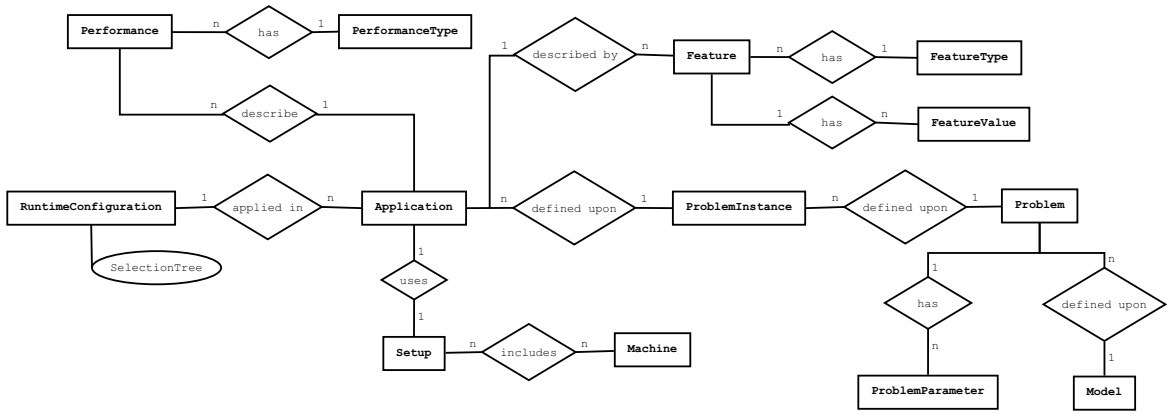

Figure 5.4.: An entity-relationship diagram of the performance database. All attributes except the selection tree stored with the runtime configuration have been left out for clarity.

5.1.2. Generality

The entity descriptions of section 5.1.1 show that some database entities are based on JAMES II concepts. However, the overall *structure* of the database, i.e., the fundamental entities and their relationships, is likely to be the same for any other modeling and simulation system. Figure 5.4 displays the *entity-relationship (ER) diagram* [271, 124 et sqq.][5] of the performance database. To transfer this layout to another simulation system, one needs to replace the following (sub-)entities, which should be straightforward:

- *Selection trees* define the concrete configuration of the simulation system that is used. While both monolithic as well as monolithic-parametric algorithms can be easily mapped onto this structure, other simulation systems may implement simpler structures that better reflect their abilities.
- *Feature extractors* are defined on JAMES II types (`ParameterBlock` and its specific content); they have to be adapted to the corresponding entities of the host system.
- *Performance measurers* are defined on `SimulationRuntimeInformation`, a JAMES II-specific type, and hence need to be adapted as well.

Finally, note that the terminology introduced in the preceding paragraphs is specific to the SASF and *not* generally established — there is no widely accepted common terminology. For example, the application of a *runtime configuration* to a *problem instance* is called a *trial* in [227].

5.1.3. Implementation Details

To ensure scalability with the amount of recorded performance data (see tab. 4.2, p. 99), the performance data storage should be realized by established methods for managing large amounts of highly structured data, i.e., a *database management system (DBMS)*. Yet, user preferences regarding the specific *kind* of DBMS may differ strongly, e.g., depending on the operating system, the amount of performance data to be collected, or financial constraints. It might be a good choice, for instance, to use a simple file format in case of a small experiment with little data to be recorded. The exploration of algorithm performance on larger sets of simulation problems and algorithms, on the other hand, may even require to set up a dedicated server for a centralized performance database, which can be accessed by many users.[6]

[5] Instead of the "crow's foot" notation described in [271], the relationship degree is represented conventionally, i.e., by edge labels: $1 - n$ stands for a one-to-many relationship, and so on.

[6] A realization of such a web-based performance database requires additional data structures, e.g., for user management.

The performance data storage of the SASF should therefore be accessed via an interface only, i.e., it should be independent of a concrete implementation. This ensures a separation between the concerns of working with the data and managing it. In contrast, other performance storage modules of algorithm selection systems are DBMS-specific, e.g., PYTHIA II relies on the POSTGRES95 DBMS [146, p. 235].

For the SASF, introducing a new plug-in type *performance database* allows to support alternative implementations. Its base factory `PerfDBFactory` provides users with implementations of the `IPerformanceDatabase` interface. Besides an (incomplete) implementation that relies upon the official *Java Database Connectivity (JDBC)* [246] interface, an implementation based on the *Hibernate* [129] persistence system has been implemented.

JDBC offers a common *interface* to various DBMS that provide suitable drivers, but its applicability suffers from the limited compatibility between different DBMS, even though they are all accessed by the *structured query language (SQL)* [271, p. 23–24]. Hibernate, in contrast, provides much more than just a common interface for sending SQL statements to a DBMS and retrieving the results [63]: it automatically maps objects to database structures and provides an intermediate language, the *Hibernate Query Language (HQL)*, to manage them. The actual structure of the database — which could be relational or object-oriented, for example — is hidden from the programmer. HQL statements will be translated to the specific SQL *dialect* of the DBMS at hand. Hence, using Hibernate results in full compatibility with all DBMS that provide Hibernate drivers, without having to change any code. The Hibernate-based implementation has so far been used in combination with the relational open-source DBMS MySQL [236, 271]. The MySQL tables for the performance database are depicted in figure A.1 (p. 222).

The structure shown in figure 5.4 (p. 111) is complemented by additional database views that aggregate performance summaries, e.g., regarding best- and worst-performing runtime configurations per model, and so on. A dedicated user interface for the performance database has not been developed yet, the database is accessed either by its API or by generic DBMS front-ends.

Several simple enhancements have been introduced to the abstract structure shown in figure 5.4, all of which aim at improving database performance. Most importantly, problem entities and runtime configuration entities have an extra attribute that represents a hash value for model parameters and selection tree, respectively. This allows to speed up the retrieval of equivalent entities.

5.2. Performance Recording & Feature Extraction

This section details how the entities defined in section 5.1.1 are recorded within JAMES II, and which components are responsible for their creation and storage. These form the monitoring component depicted in figure 5.1 (p. 101).

Feature Extraction

The actual extraction of features, i.e., the application of feature extractors to all applications stored in the performance database, is handled by a generic `FeatureExtraction` class. Feature extraction does not have to be triggered during the execution of an experiment, since all data on the simulation problem is stored within the performance database. The model can be instantiated with exactly the same structure, i.e., by using the same parameters and also the same random number generator with the same seed. This allows to add *new* feature values at any later time, given that the models are still located at the URIs stored in the database (see sec. 5.1.1, p. 103). The application of feature extractors is often restricted to specific model formalisms, so that feature extractor factories rely on the `IParameterFilterFactory` interface to control for what kind of application entity they are used (see sec. 4.2.3, p. 87). The database ensures that no application is associated with two features of the same type, so that this procedure can be repeated whenever new applications or feature types are introduced, without storing any redundant information.

Performance Recorder

In contrast to features, which can be extracted from re-instantiated simulation problems at a later time, many important performance metrics, e.g., execution time or memory consumption, have to be measured *during* a performance experiment and need to be stored afterwards. The performance measurers introduced in section 5.1.1 (p. 108) are hence invoked by an additional component, the `PerfDBRecorder`. It applies the measurers in the same way `FeatureExtraction` handles feature extractors: each measurer factory has to decide whether or not the measurer can be used for the given application entity, again by implementing the `IParameterFilterFactory` interface. All suitable measurers are called by `PerfDBRecorder`. Their results are stored in the performance database.

However, before associating performance measures with an application, the application entity itself has to be created and stored — and with it all entities on which it relies, i.e., model, problem instance, runtime configuration, setup, and so on. This procedure is largely automated and managed by the `PerfDBRecorder` as well.

At first, model and problem entities are generated and stored to the database, which only adds them in case they are unique — otherwise the entity already present in the database is returned to the recorder. The information they contain can be easily retrieved from the simulation job descriptions used within the experimentation layer of JAMES II, to which the `PerfDBRecorder` is attached (see sec. 7.1). To support the use of common random numbers (see sec. 3.2.1, p. 66), the `PerfDBRecorder` is equipped with a `ProblemInstanceGenerator` that either re-uses existing problem instances already stored within the database or creates a new one. The number of problem instance entities can be configured by the user.

Application entities link the problem instance at hand with a given hardware setup and a runtime configuration. The description of the hardware setup is rather informal, mostly containing free-format text fields to convey information. This is hard to automate, so entering the setups is currently left to the user. If no setup is specified, a default one will be created — this should be unproblematic, as long as all collected data come from the same hardware setup.

The most challenging entity to record, however, is the runtime configuration. It contains a selection tree for JAMES II, which might be quite large. Entering it manually would be cumbersome. Furthermore, the default semi-automatic selection process implemented in the JAMES II registry (see sec. 4.2.1, p. 85) does *not* ensure that a predefined setup is actually used: if one algorithm is not available or its factory declines its suitability for the given problem, the registry will replace it by an alternative implementation. While this behavior increases the robustness of JAMES II, it also makes it harder to determine which algorithms (and which parameters) have actually been used. Selection trees are therefore constructed automatically by a `SelectionHook`, which is passed to the JAMES II registry by the `PerfDBRecorder`. The JAMES II registry notifies this hook whenever the `getFactory(...)` method of the registry is called (see sec. 4.2.1, p. 84). The selection hook then analyzes the current thread's stack trace and adds a node that defines which factory has been selected with which parameters. If the performance of an executed simulation run shall be stored, i.e., the execution is finished, the `PerfDBRecorder` queries the selection hook to retrieve the complete selection tree for the given thread.

Current Limitations

The current approach of recording performance data is limited in some important ways. It presumes that:

- All factories are chosen with the `getFactory(...)` method.
- All factories are chosen within a single thread.

Note that these limitations do *not* restrict the general applicability of the overall system. The first limitation is controlled by the developer, who has to refrain from hard-coded factory selection. The second limitation refers to factory *selection*, not execution: after all factories have been selected via

`getFactory(...)`, they *still* may create arbitrary new threads, even on remote machines. The only thing these other threads should not do is to call `getFactory(...)` from their local registry, which will not have the same selection hook installed.[7] Of course, one can easily think of more elaborate alternatives to record selection trees, potentially allowing to control remote selection processes or even ensuring that *all* managed algorithms are captured. These mechanisms could be realized, for example, by using Java's reflection API [245] or applying advanced programming paradigms like aspect-oriented programming, which has already been proven useful to implement other cross-cutting concerns in simulation software [50]. However, implementing a more sophisticated recording mechanism requires major structural changes to the registry and will implant performance recording in a much more intrusive manner — currently, no selection information will be gathered unless the performance shall indeed be recorded.

Finally, the possibilities of performance measurers are limited as well: not by the current implementation of the performance recording, but rather by the host system itself. In case of JAMES II, performance measurers have access to simulation output (via implementations of `ISimDataProvider` contained in application entities, see sec. 5.1.1, p. 106) and all information delivered to them by the experimentation layer. Measures defined upon observations *during* execution, e.g., the number of rollbacks in an optimistic PDES execution, currently require additional efforts to integrate. The simplest way to realize such measures is to record them together with ordinary simulation output, and then to retrieve the measurements by the application's `ISimDataProvider` instance.

5.3. Summary

This chapter introduced the part of the simulation algorithm selection framework that is responsible for managing and collecting performance data. It is largely built upon the conceptual entities proposed by Rice (see sec. 2.1, p. 13), but now decomposed into concrete entities that are specific to modeling and simulation.

The overall structure of the database — the entities, their relation, and the content to be stored in each of them — is *generic* and can be adopted by any other simulation system. Specifics of the host system have to be reflected, nevertheless: the description of algorithms, mechanisms to extract features from simulation problems, and mechanisms to measure performance (sec. 5.1.2, p. 111). Furthermore, the recording components themselves (sec. 5.2, p. 112) are highly system-specific and have only been developed as prototypes.

Figure 5.5 summarizes the most important software entities discussed in this chapter. All the data that can now be stored is yet to be analyzed, so that it can be eventually used to the create selection mappings that solve the ASP. The next chapter describes how this can be done.

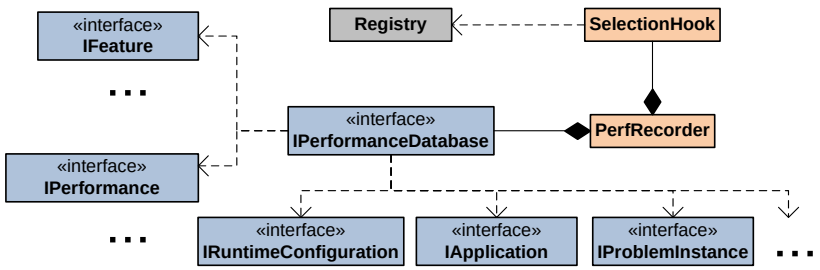

Figure 5.5.: UML diagram of the most relevant components for data recording and storage in the SASF. For full flexibility, all concrete implementations of performance data entities are hidden by interface definitions. The colors again denote different parts of the SASF: data storage (blue) and monitoring elements (orange), see figure 4.6 (p. 97).

[7] Or, in case multiple threads run within the same JVM, this will cause the selection hook to presume that this is a new simulation run, with a new selection tree.

6. Selection Mapping Generation

> *[...] as the complexity of a system increases, our ability to make precise and yet significant statements about its behavior diminishes until a threshold is reached beyond which precision and significance (or relevance) become almost mutually exclusive characteristics.*
>
> Lotfi Zadeh, from [339, p. 28]

Chapter 2 already illustrated why only an empirical approach to algorithm selection is likely to succeed in the foreseeable future. Now that monitoring mechanisms are in place to feed observations into a dedicated performance database, as described in chapter 5, it has to be discussed how this data can be analyzed so that suitable selection mappings are generated (see fig. 6.1).

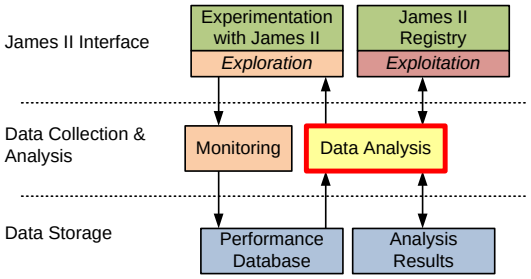

Figure 6.1.: SASF overview (see fig. 4.6, p. 97), red borders denote elements discussed in this chapter.

Many concrete techniques for constructing selection mappings stem from the field of machine learning, which has already been related to the ASP in section 2.3 (p. 24). These techniques employ inductive reasoning, i.e., they extrapolate past algorithm performance to future problems by constructing approximation forms — the selection mappings to solve the ASP (see def. 2.1.2, p. 15). In other words, the methods are susceptible to the pitfalls and problems detailed in section 1.4 (p. 8) and hence have to be evaluated carefully.

The form of selection mapping has been identified to be an important ASP aspect right from the beginning; Rice considers it to be essential and envisions a variant that strongly resembles a decision tree [272, p. 94]. Since machine learning aims at generating selection mappings, it can be regarded as a *solution technique* for the ASP, and is discussed as such in section 2.6.1 (p. 50). Section 6.1 gives a more detailed overview on concrete machine learning techniques that have been applied to the algorithm selection problem and derives some more specific requirements from their commonalities and differences. Then, section 6.2 details the parts of the simulation algorithm selection framework that realize a similar functionality. The main ideas described in this chapter have been presented in [76]. Note that this chapter is focused on *supervised* learning: learning from past performance data that is stored in the performance database, observed on problems that have been fully explored, i.e., it is clear which algorithm actually performed best. The application of unsupervised methods is discussed in section 7.2.

6.1. Learning Algorithm Selection Mappings

Machine learning — or its more data-intensive and application-focused sibling science, data mining — consists of various sub-fields. The documentation of the Java data mining standard JDM speaks of classification, regression, attribute importance, clustering, and association as the main functions of data mining [167, p. 6]. While all of them may be important for an algorithm performance analyst, section 2.3 (p. 24) shows that the ASP can be best mapped to a learning problem.[1] A problem

[1] The feature selection problem (def. 2.1.3, p. 15), in turn, is concerned with estimating the attribute importance with respect to the ASP.

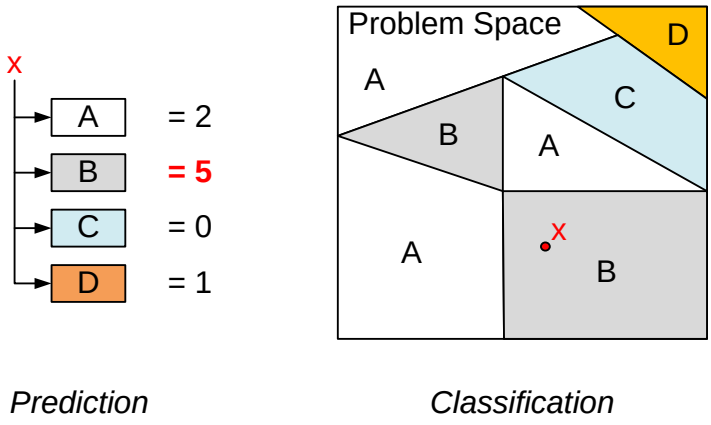

Figure 6.2.: *Performance prediction* vs. *problem classification*. On the left, a predictor is confronted with the features of the simulation problem $x \in \mathbb{P}$, and also with the configuration to be used $(A, \ldots, D)$. It is called with each algorithm to predict its performance when applied to x. The algorithm with the best predicted performance is chosen (B). A classification approach (right) decides in which region of the problem space a given simulation problem lies, i.e., which algorithm is deemed best for the specific region (again, B).

classification approach to solve the ASP chooses the best algorithm from a finite set by considering the problem features, whereas a prediction approach predicts each algorithm's performance (from the infinite set $\mathbb{R}$) and then chooses the algorithm with the best performance (see sec. 2.6.1, p. 51). Both approaches are outlined in figure 6.2.

While there is a multitude of methods described in introductory literature (e.g., [122, 144, 333]), a closer look at the methods used by other ASP solution approaches may hint at the most promising directions to investigate.

The first question is whether to follow a performance prediction or a problem classification approach. This decision is also discussed in the literature (see sec. 2.6.1, p. 51). One issue raised in [205], for example, concerns the adaptation of a selection mapping when a new algorithm is introduced: while prediction just requires to learn a new prediction model for the new algorithm (e.g., see eq. 2.13, p. 25), classifying problems by the most suitable algorithm requires to reconfigure the *whole* classifier, i.e., *all* past performance data have to be reconsidered in some form.

Note that, as Leyton-Brown et al. point out in [205, p. 36–37], it is also possible to use a classifier[2] for the performance prediction approach, by classifying an algorithm's performance on a given problem as `GOOD`, `MEDIUM`, or `BAD`. While such kinds of classifications do not necessarily need overall adaptation to cope with new algorithms, it might be that the qualitative notion of what is considered to be `GOOD` performance still changes and hence results in additional efforts. The confrontation with newly developed algorithms is likely to be quite common when relying on on a host system as open and flexible as JAMES II — each newly developed sub-algorithm will result in one or more 'new' runtime configurations and hence elements in $\mathbb{A}$. This suggests to use regression methods for prediction.

Many other approaches to solve the ASP by classification instead of prediction. Table 6.1 gives a brief summary of some concrete techniques that have been used. Apparently there is no generally accepted machine learning method to be used. The reasons for this are the differing problem domains and the bias-variance trade-off described in section 2.3.1 (p. 25): the performance of machine learning methods is highly problem-dependent, hence a careful application of *multiple* methods seems mandatory. Both decision trees (for classification) and linear regression (for prediction) have been used quite often. This is because they are rather simple to implement and use, and also because their structure can be interpreted easily by humans — which makes them suitable not only for generating selection

[2]Here, the term classifier refers to classification in the sense of machine learning (sec. 2.3, p. 24), *not* to the problem classification approach depicted in fig. 6.2.

Domain	Prediction vs. Classification	Technique	Used in, e.g.,
Sorting	P	Linear regression	[25]
Combinatorial Search (Winner Determination)	P	Ridge-Regression, Support Vector Machines, Multivariate Regression	[205]
Combinatorial Optimization (Most Probable Explanation)	C	Decision Trees, Naïve Bayes, etc.	[117]
Constraint Satisfaction	C	Bayesian Networks, Decision Trees	[145]
Meta-Learning	C	Decision Trees, Naïve Bayes, Nearest-Neighbor, etc.	[257]
PYTHIA II	C	Decision Trees, Rule Induction	[146]
Matrix Multiplication	P&C	Linear Regression, Support Vector Machines, etc.	[319]
Scientific Kernels	P&C	Linear Regression, Decision Trees	[338]

Table 6.1.: Machine learning techniques for solving the ASP. Most approaches used several learning methods, some even mix performance prediction and problem classification approaches.

mappings, but also to analyze performance. Finally, it is interesting that problem classification and performance prediction can also be used in combination — they are not mutually exclusive.

In their formal framework for algorithm selection [319], for example, Vuduc et al. distinguish between *data modeling* and *geometric modeling*. Data modeling means to predict the performance of a single implementation, whereas geometric modeling aims at partitioning the feature space into regions that are dominated by a single algorithm. These concepts are equivalent to the notions of prediction and classification, as depicted in figure 6.2 (p. 116).

After considering related efforts, it seems evident that any *general* architecture to generate selection mappings for simulation algorithms should allow the integration of *various* methods, both for prediction and for classification. Since different methods are to be supported, it is likely that the methods required for their evaluation are differing as well. All these aspects again suggest the design of a *framework* into which several techniques can be plugged. It can be based upon the mechanisms already provided by JAMES II. Flexible support of learning methods is already featured by several other approaches and systems, e.g., this is also claimed in [97, 146, 205].

So far, machine learning has only been applied seldom to simulation performance analysis. Notable exceptions are the work of Ferscha et al., who mine performance data from parallel and distributed discrete-event simulations (see sec. 3.3.2, p. 73), as well as Xu and Tropper [337], who use nearest-neighbor classification to determine whether a sequential or a parallel simulator shall be used. Finally, note that the machine learning methods given in table 6.1 are not necessarily the end of the rope. Requirements may change with the desired characteristics of an ASP solution, e.g., when solving the problem at processing time (see sec. 2.6.1, p. 50). The above techniques have all been used to analyze *past* performance data, e.g., imported from a performance database, *before* processing time. ASP solution approaches at runtime may introduce different supervised machine learning methods, e.g., in [93] Gagliolo and Schmidhuber use an artificial neural network to train their adaptive mechanism (see sec. 2.5.3, p. 42) in-between the processing of problems. Recent work even showed how methods like *symbolic regression* allow to identify invariants, i.e., natural laws, in large sets of input data [283]. Such methods could be applied to find 'natural laws' in a given set of algorithms and — if such exist — use them to generate a good selection mapping.

6.2. A Framework for Simulator Performance Data Mining

As section 6.1 motivates, it is necessary to offer various methods to generate and evaluate selection mappings. A selection mapping is merely a formal notion so far (see def. 2.1.2, p. 15); it can come in various implicit or explicit forms. For example, the current JAMES II setup as described in section 4.2.1 (p. 4.2.1) already offers a form of (static) algorithm selection, i.e., it already implements a selection mapping that is implicitly defined by the efficiency indices, which in turn are maintained by the available factories.

To distinguish the theoretical notion of a selection mapping from a software component that implements it, the latter will be called (algorithm) *selector* in the following. The next section describes how selectors for JAMES II can be generated, and how they can be constructed to support both prediction and classification. Then, section 6.2.2 discusses how selectors can be evaluated and section 6.2.3 details how this newly developed *simulator performance data mining framework (SPDM)* is integrated into the overall layout of the SASF. The general procedure implemented by the SPDM is sketched in figure 6.3.

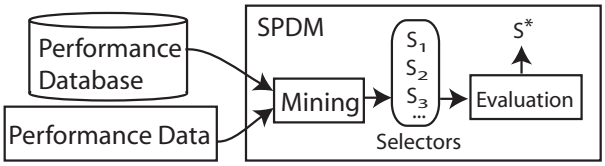

Figure 6.3.: Overall SPDM workflow. Performance data can be imported from files or the performance database. Selectors are generated, evaluated, and eventually the best one (S^*) should be used.

6.2.1. Selector Generation

The generation of selectors by considering past performance data is assigned to components of a new *selector generator* plug-in type. The performance data is passed to a selector generator in the form of a `PerformanceDataSet`, which is defined along the lines of definition 2.3.1 (p. 24). It contains the performance data and some meta-data. The performance data is represented by a list of `PerformanceTuple` instances (def. 2.3.1, p. 24). A `PerformanceTuple` contains two main sub-structures: an instance of `Features` that stores all $(name, value)$ pairs of features that could be extracted from a simulation problem instance (see sec. 5.1.1, p. 106), as well as an instance of `Configuration` that represents the runtime configuration (see sec. 5.1.1, p. 104). Besides these elements from $\mathbb{F}$ and $\mathbb{A}$, a performance tuple contains a floating-point number $p \in \mathbb{R}$ denoting the performance of the configuration on a problem with the given features. Finally, it stores a reference to the class of the performance measurer factory (see sec. 5.1.1, p. 108), so that it is clear what *kind* of performance it refers to.

The meta-data is given by an instance of `PerfTupleMetaData`, which keeps information on numerical and nominal attributes for easier reference. It facilitates the integration of external methods.

Selection Tree Flattening

As described in section 5.1.1 (p. 104), simulation algorithms in JAMES II can be represented by selection trees, which are part of the runtime configuration entities (sec. 5.1.1, p. 106) and specify all sub-components used for execution (including their parameters). Yet, methods for machine learning usually consider the *attributes* of an instance as a set of $(name, value)$-tuples.

Therefore, it is necessary to transform the selection tree of a runtime configuration to a unique set of $(name, value)$-tuples, in order to ensure that all setups can be distinguished by the learning method. This is done by flattening the selection tree with an additional component, the `SelectionTreeFlattener`. The result of this flattening procedure, i.e., the set of $(name, value)$-tuples that specifies a runtime configuration of JAMES II, is simply called *configuration* in the following.

The flattener traverses a selection tree and stores all selection information to a mapping $String \rightarrow Object$, i.e., a structure similar to the simulation problem features (sec. 5.1.1, p. 106). Only by

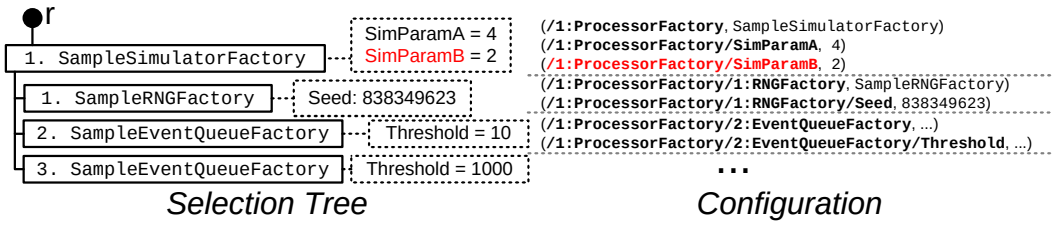

Figure 6.4.: Selection tree flattening. This figure shows how the sample selection tree from figure 5.3 (p. 105) is translated into a unique set of $(name, value)$ tuples. For example, the value for `SimParamB` (red) is associated with a name that corresponds to the path from the root, each vertex represented by its numbering and its plug-in type (i.e., base factory; here: `ProcessorFactory`). Parameter names are appended at the end. All other configuration choices are stored similarly. Configuration aspects that are deemed negligible (e.g., the seed of the RNG) can be filtered out before selector generation.

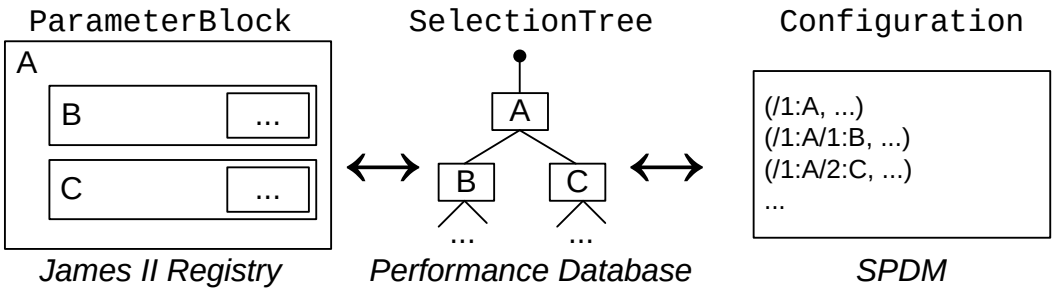

Figure 6.5.: The relation between `ParameterBlock`, `SelectionTree`, and `Configuration` instances and their usage domains. `ParameterBlock` instances are used to query the registry; they do not distinguish between selected factories and their parameters. `Configuration` instances are used for machine learning and hence provide the same data in a flat structure (cf. fig. 6.4).

combining problem features and configuration features it can be assured that the learner learns the relation between problem features and performance for the *right* algorithm. The flattening procedure as such has to ensure that its results are unique for any selection tree. This is done by using the *path* from the selection tree root r (def. 5.1.1, p. 105) to each node as key, and setting the factory represented by this node as the attribute value. Parameters are handled similarly. The general idea is illustrated in figure 6.4.

The numbering scheme introduced in definition 5.1.1 (p. 105) ensures the unambiguity of the resulting tuples in case multiple sub-components of the same type — e.g., event queues — are used by the same component (see fig. 6.4). This ensures that the corresponding selection tree can be recreated, based on the flattened configuration attributes. Such reversion, however, is only of theoretical interest: in practice it seems best to create the $(name, value)$ tuples required for machine learning while *keeping* the corresponding selection tree. In case a given configuration is deemed best, its selection tree can later be used to generate the parameter block that configures JAMES II correspondingly. The relation between `ParameterBlock`, `SelectionTree`, and `Configuration` is sketched in figure 6.5.

Selectors & Performance Predictors

The formal definition of a selection mapping suggests a very simple selector interface: given some features extracted from the simulation problem at hand, it should simply return the most suitable configuration. Yet, in case of JAMES II and any other *flexible* host system,it cannot be assured that the configuration picked by the selector is available to the user. This additional uncertainty can be encountered by redefining the role of a selector. One may regard it as a *sorting* mechanism for runtime configurations.

It works in analogy to the implicit algorithm selection mechanism that is already integrated in the factory filtering process of the JAMES II registry (see sec. 4.2.1, p. 85). This means a selector does not simply *pick* the best configuration, but instead it *sorts* all configurations it knows by their predicted performance: the algorithm that is deemed best is the first element in the list, the second one is on position two, and so on. This allows to select the best *available* setup, and not just fail if the user did not install a certain plug-in.

The Java Collections API provides a generic sorting mechanism that can be used by implementing the `Comparator` interface. A comparator just has to provide a single `compare(T` o_1`, T` o_2`)` method that takes two objects o_1, o_2 of type T as parameters and returns an integer lesser than, equal to, or greater than 0 to distinguish between $o_1 < o_2$, $o_1 = o_2$, and $o_1 > o_2$. This makes it possible to implement a generic selector class that merely takes alternative implementations of `Comparator` to adapt its behavior. Figure 6.6 outlines the actual implementation: the `ISelector` interface is responsible for sorting configurations by considering the features of the simulation problem at hand. It does so by using an instance of `IPerfComparisonPredictor`, which in turn extends the generic Java interface `Comparator`. As the `IPerfComparisonPredictor` controls the overall sorting of the configurations, it is the only interface that needs to be implemented when adding a new kind of selector. Due to the use of `Comparator`, the sorting task is transformed into a *sequence* of decisions between only two algorithms. Either one of them is predicted to be faster than the other ($o_1 < o_2$, $o_1 > o_2$), or the mapping does not distinguish between the performance of both algorithms on the given problem ($o_1 = o_2$).

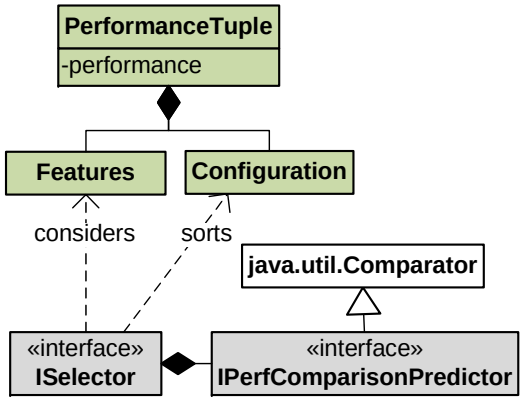

Figure 6.6.: Basic SPDM entities for selector generation.

Classification and prediction methods may still approach the problem in very different ways, although their selectors ultimately implement the same interface. For example, a problem classification approach could aim to classify tuples of the form

$$(f, a_1, a_2) \in \mathbb{F} \times \mathbb{A}^2 \tag{6.1}$$

into the categories `firstBetter`, `secondBetter`, or `equal`. This means that for each comparison a single prediction has to be made, namely how the performances of the two algorithms relate to each other. A drawback of this approach is that the number of tuples as given in 6.1 grows *quadratically* with the number of algorithms — which may result in scalability problems that should be avoided (see tab. 4.2, p. 99).

In contrast, a prediction approach would predict the performance of *each* algorithm $a_1, a_2 \in \mathbb{A}$ in isolation, with a predictor $p_{pred} : \mathbb{F} \times \mathbb{A} \to \mathbb{R}$, and then just compare the predicted performances $p_{pred}(f, a_1)$ and $p_{pred}(f, a_2)$. As discussed in section 6.1, this could be realized with classification[3] as well as with regression. The former requires some additional efforts in case new algorithms are added, as the qualitative assessment of an algorithm's performance (e.g., `GOOD` and `BAD`, see sec. 6.1, p. 115) is relative to the performance of the others. Since the performance space defined in the ASP is real-valued (see def. 2.1.2, p. 15), using regression for the performance prediction approach seems like a natural fit. Note that predicting the performance of *each* algorithm and then comparing the outcomes requires two (instead of one) predictions per comparison. This may hamper the scalability of the approach in case the prediction procedure requires some calculation, but can be easily circumvented

[3] Again, this refers to classification in the sense of machine learning (sec. 2.3, p. 24), not the algorithm selection approach of problem classification (fig. 6.2, p. 116).

by *caching* the predicted performance per algorithm and re-using it for all comparisons it is involved in.

To avoid the scalability problems that come with the problem classification mechanism, all currently implemented selector generation schemes follow the prediction approach outlined in figure 6.2 (p. 116). Still, problem classification approaches can be easily realized within SPDM as well.

Integration of External Tools

The SPDM is designed as a framework. It is advisable to rely on external libraries for selector generation, instead of re-inventing the wheel by re-implementing all (potentially complex) machine learning methods that could prove beneficial. The following paragraphs briefly describe which tools have been integrated so far, and which approximation models are used (i.e., which form of selection mapping is supported, see sec. 2.3.1, p. 25).

Weka WEKA [327, 333] is a powerful open-source software that is written in Java and provides various algorithms for machine learning. They are accessible either by a graphical user interface or via an API. Since decision trees have already been used to solve the ASP (see tab. 6.1, p. 117), a selector generator based on WEKA's version of the C4.5 decision tree algorithm, `J48`, has been implemented [333, p. 159 et sqq.].

Decision trees are trees that have attribute names as nodes, potential attribute values (or ranges, in case of numerical attributes) as edges, and the prediction outcome as leaf nodes. As outlined in figure 6.7 (p. 122), they are evaluated top-down: for each node, the corresponding attribute of the problem instance in question is considered. The edge that matches the instance's attribute value is followed to the next node. If a leaf is reached, the decision is returned. Decision trees can be constructed by considering the Shannon entropy [333, p. 93–94] of past data with respect to each attribute (also see sec. 2.4.1, p. 34). Thereby, the attribute with most information gain, i.e., the knowledge of which most strongly determines the class of an instance, is chosen first (and so on).[4] This simple structure — more important attributes higher in the tree than less important ones — does not only help manual interpretation by a performance analyst, but also reduces the number of features that have to be calculated for classification. In the example given in figure 6.7, only the features `Rain?` and `Windy?` have to be measured, while the value of `Cold?` can be neglected — it is not necessary to know the value of `Cold?` for classification, as long as the attribute `Rain?` has the value 'No'. Since extracting problem features (such as `Cold?`) could be a time-consuming task as well, requiring only those attribute values that truly matter in the given situation is an advantage of decision trees.

To create a decision tree that realizes the prediction approach as outlined in figure 6.2 (p. 116), the past performances are assigned to n distinct performance classes, where n is a parameter: class 1 is considered best and class n is considered worst. Given the features of a new simulation problem, the decision tree predicts the performance class for each configuration. This class is then used to sort the configuration entries. The process is depicted in figure 6.8 (p. 122).

Furthermore, WEKA offers the construction of so-called *model trees*, which extend decision trees towards regression by storing a linear regression model at each leaf node [333, p. 201 et sqq.]. In other words, the decision tree structure is not used for predicting a value anymore, but for predicting which linear regression model provides the best (numeric) prediction. Model trees have been integrated by using WEKA's M5P learner, which is based on the M5 algorithm from [269]. Here, no preprocessing of performance tuples, i.e., assigning them to n performance classes, is necessary.

MLJ MLJ stands for *Machine Learning in Java* [214]. The library is basically a Java-port of MLC++ [184], a framework that aims at comparability and repeatability of machine learning experiments, in a similar way that JAMES II aims at comparability and repeatability of simulation

[4] By doing so, the learning scheme implicitly attempts to solve the best features for algorithm problem (def. 2.1.3, p. 15), i.e., it identifies the most relevant problem features.

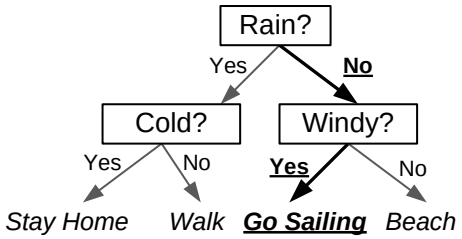

Figure 6.7.: A decision tree example. First, the attribute `Rain?` is checked (no, bold), then the attribute `Windy?` (yes). Afterwards a leaf is reached: the tree decides for 'go sailing'.

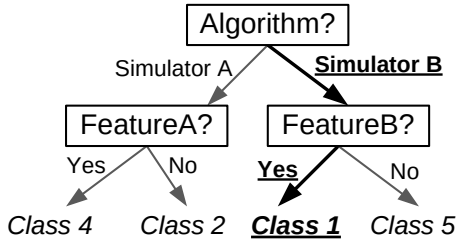

Figure 6.8.: Using a decision tree to solve the ASP. Attributes that relate to features (`FeatureA?`, `FeatureB?`) are not distinguished from those that relate to the configuration (`Algorithm?`). The latter are derived from the flattened slection trees (see fig. 6.4, p. 119).

experiments. A wrapper for its ID3 implementation, another algorithm to construct decision trees (see fig. 6.7), was added to SPDM, as well as a *Naïve Bayes* classifier (e.g., [333, p. 82]).

It is based on Bayes' rule of *conditional probability*:

$$Pr(A|B) = \frac{Pr(B|A) \cdot Pr(A)}{Pr(B)} \tag{6.2}$$

$Pr(A|B)$ denotes the probability of event A given that event B occurred. It can be calculated by the probability of event B occurring under the condition that A occurred ($Pr(B|A)$) and the occurrence probabilities of A and B ($Pr(A)$ and $Pr(B)$, respectively). For classification, B represents the attributes of the problem instance to be classified and A represents the class to be predicted. The probabilities $Pr(B|A)$, $Pr(A)$, and $Pr(B)$ are then estimated from past performance data by counting: how often have the attributes in B occurred if an instance belongs to class A ($Pr(B|A)$), how often did class A occur in general ($Pr(A)$), and how often did the attributes in B occur in general ($Pr(B)$). The calculation of equation 6.2 is repeated for each class; the class with maximal probability is chosen as classification result.[5] Note that the above procedure is called naïve because $Pr(B|A)$ is estimated by merely multiplying the occurrence probabilities of each attribute given A, hence assuming the attributes to be *independent* of each other. This assumption is often wrong. For example, a simulator and its event queue would be considered independent, even though the efficiency of the event queue could strongly depend on the usage pattern of the simulator. In contrast to decision trees, this classifier also treats all attributes as being equally important — this would be another reason to call it naïve. Nevertheless this simple scheme has proven to be quite useful in many scenarios [333, p. 88], e.g., to filter spam mail.

Joone JOONE is an engine for (artificial) neural networks, written in Java [216]. Neural networks are restricted to real-valued input, so that all nominal attributes (e.g., the algorithm being used) need to be encoded into real-valued input within the wrapper for SPDM. The functioning of neural networks as a method for regression has been inspired by processes observed in neurobiology. A *neural network* can be regarded as a weighted directed graph that is divided into different layers $L_1, \ldots, L_n$. All edges go from layer L_i to layer L_{i+1}.

At first, the real-valued attributes of a problem instance are fed into the nodes of the input layer (L_1). A *transfer function* is defined for each node (usually the same); it transforms input values to output values. The output of a neuron's transfer function gets propagated along all outgoing edges to the neurons of the next layer. This is done for all neurons in the layer. Then, each neuron in the next layer aggregates the input it received from all incoming edges and interprets it as its input, i.e.,

[5] Additional considerations help to avoid situations where $Pr(A)$ or $Pr(B)$ are zero [333, p. 82 et sqq.].

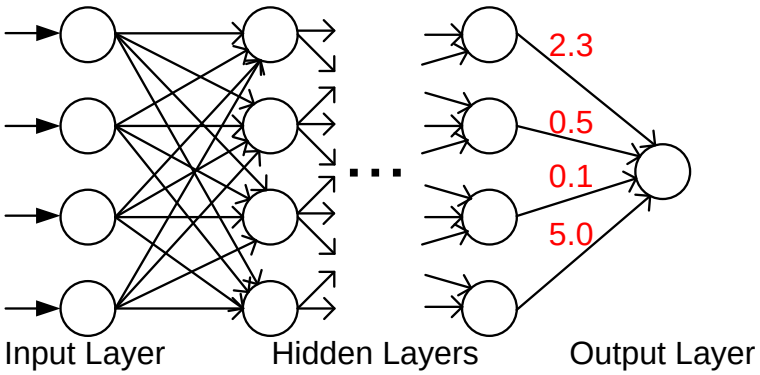

Figure 6.9.: Basic outline of a neural network. Input is propagated from the input layer over the hidden layers to the output layer. Edge weights have been omitted for clarity, except for the single neuron of the output layer (red).

stimulus. It applies its transfer function to generate output that is propagated to the next layer, and so on.

All this reflects the basic functioning of their biological counterparts, where neurons propagate signals via axons, synapses, and dendrites. The main idea is that biological neural networks are able to learn by adjusting the degree of signal propagation from one neuron to another. Artificial neural networks can imitate this by changing the edge *weights* by some specific learning scheme, e.g., back-propagation [275, p. 151 et sqq.]. Figure 6.9 illustrates their basic structure.

Joone is highly configurable, and so is its SPDM wrapper. Besides the size of the model, it is possible to switch between different transfer functions and learning algorithms.

JDM Finally, first steps towards the integration of the Java data mining standard JDM [167] have been undertaken. Though being a standard and hence the *only* interface SPDM would have to support in an ideal world, there is little support for it, particularly in terms of open-source implementations. The current implementation of a JDM interface for SPDM was briefly tested with a commercial toolkit developed by KXen, Inc. [186], but was eventually dismissed due to lack of open-source alternatives. The lack of momentum behind JDM also becomes apparent when considering the efforts towards a new JDM version [166], which date back to 2006.

6.2.2. Selector Evaluation

The selector generators described in section 6.2.1 have several options to adjust them to the given problem. Anyhow, it is usually not clear a priori which machine learning configuration is best;[6] one reason for this is the bias-variance trade-off discussed in section 2.3.1 (p. 25).

It thus seems reasonable to let the SPDM automatically evaluate the performance of *multiple* selector generators, on the same set of input data, to see which generated selector works best. It usually takes considerable time to collect the required performance data. Since constructing new selection mappings is usually much faster than performance data collection, restricting the data analysis efforts of the SPDM would not save significant amounts of computation time — but it could impair the ASP solution quality.

To provide a fair evaluation of the generated selectors, these have to be compared with the *same* methods. The evaluation components have been realized within the SPDM itself. This ensures comparability and also allows to further structure the problem. The quality of a selector can be expressed by its expected prediction error, which can be estimated by applying it to formerly *unseen*

[6] This is the problem meta-learning tries to solve; as discussed in section 2.7 (p. 58) it can be used to manage generated selection mappings, or to decide which learning algorithm to use.

test instances (see sec. 2.3.1, p. 25). Hence, two distinct components are required for a performance evaluation: firstly, a *selector performance measurement*[7] that quantifies the error of a selector on a test set. Secondly, an *evaluation strategy* to divide the overall data into test and training set. Both tasks are implemented by JAMES II plug-in types; their rationale and current implementations will be briefly described in the following.

Evaluation Strategies

Hastie et al. recommend the usage of *three* distinct tuple sets [122, p. 196] if enough data is available: one for training, one for testing, and one for eventually validating the selector that performed best on the test set. While they maintain that set sizes depend on the approximation model that shall be learned (see sec. 2.3.1, p. 25), they describe a typical approach to use 50% of the data for training, splitting the remaining instances equally for testing (all potential solutions) and validating (the best one).

For solving the ASP, this would mean that only *half* of the data is actually used for selector generation — even though performance data is hard to come by and obtaining it may be computationally costly (see ch. 3, p. 61). Fortunately, there are strategies that allow a proper evaluation of selectors without the presence of an additional validation set, so that more data can be used for selector generation. Two of these widely used strategies have been been implemented for SPDM: 0.632 bootstrapping and cross-validation. They are realized as plug-ins and can hence be exchanged. Instead of splitting the performance data into three sets, they only divide it into training and test set. These are denoted by Φ_{train} and Φ_{test} in the following (def. 2.3.1, p. 24). Both sets have to be disjunct: $\Phi_{train} \cap \Phi_{test} = \emptyset$.

Since the basic idea behind having a test set is to confront the learned selector with formerly *unseen* data, it seems reasonable to split the performance tuple set Φ on the level of features, not on the level of single performance tuples. This ensures that the evaluation tests selector quality with 'new' *problems*, not just with new combinations of algorithm and problem.

More formally, let $\mathbb{F}_\Phi = \{f|(f,a,p) \in \Phi\}$ be the set of all distinct problem features stored in the performance tuple set Φ, and let $\Phi^f = \{\phi_i|\phi_i = (f_i,a_i,p_i) \in \Phi \wedge f_i = f\}$ be the set of all performance tuples in the performance tuple set that contain the same features f. Then, an SPDM evaluation strategy needs to split $\mathbb{F}_\Phi$ into two distinct sets $\mathbb{F}_{train}$ and $\mathbb{F}_{test}$, to construct Φ_{train} and Φ_{test} correspondingly:

$$\Phi_{train} = \bigcup_{f \in \mathbb{F}_{train}} \Phi^f$$
$$\Phi_{test} = \bigcup_{f \in \mathbb{F}_{test}} \Phi^f$$

Bootstrapping In statistics, the term *bootstrapping* refers to resampling techniques that allow to estimate some properties of a selection mapping, e.g., its prediction error. Resampling means to draw new samples from a larger data sample. For the given problem, it implies to select those performance tuples from Φ that make a good training set, while still being able to evaluate the generated selectors with sufficient test data. A common method is the so-called 0.632 *bootstrap* [333, p. 128], which has been implemented as an SPDM plug-in.

The idea behind it is straightforward: at first, an empty multi-set for training data is defined, $\mathbb{F}_{train}$. $\mathbb{F}_{train}$ is now filled by randomly drawing features from $\mathbb{F}_\Phi$. Note that features are drawn randomly *with replacement*, i.e., they are *not* removed from $\mathbb{F}_\Phi$ and may therefore be added to $\mathbb{F}_{train}$ more than once (hence $\mathbb{F}_{train}$ needs to be a multi-set). This process continues until $\mathbb{F}_{train}$ has the size of the original set: $|\mathbb{F}_{train}| = |\mathbb{F}_\Phi|$.

[7] These measures relate to the performance of a selection mapping, they should not be confused with the performance measures for algorithms in $\mathbb{A}$, which have been described in section 5.1.1 (p. 108).

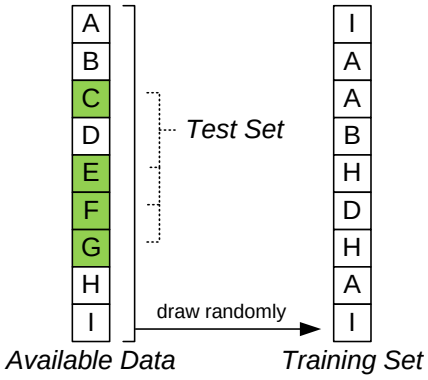

Figure 6.10.: Bootstrapping: instances are drawn randomly, the rest is used as test set.

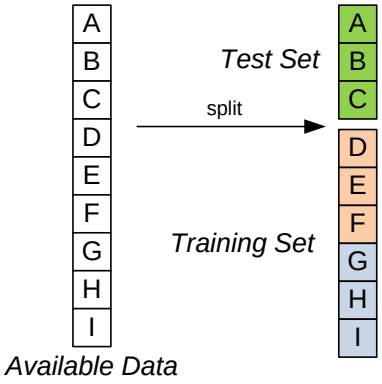

Figure 6.11.: Cross-validation: instances are split into equalls sized groups (denoted by color). One is used as test set.

Assuming $|\mathbb{F}_\Phi| = n$, each element of $\mathbb{F}_\Phi$ is randomly drawn with a probability of $\frac{1}{n}$ per round, i.e., it is *not* drawn with a probability of $1 - \frac{1}{n}$. Since there are n iterations, the overall probability that a certain element of $\mathbb{F}_\Phi$ has not been chosen at all — and may therefore be used for testing — is $(1 - \frac{1}{n})^n$. Since $\lim_{n\to\infty}(1 - \frac{1}{n})^n = e^{-1} \approx 0.368$, the test set contains about 36.8% of the overall data, hence the method is called $1 - 0.368 = 0.632$ bootstrapping. The 0.632 bootstrap draws *random* samples for testing and training, so the overall process has to be repeated several times. Figure 6.10 summarizes the overall procedure.

Cross-Validation There are several variations of *cross-validation* [333, p. 125], but the basic principle is very simple: the set $\mathbb{F}_\Phi$ is split into x pairwise disjunct sub-sets $\mathbb{F}_{\Phi,1}, \ldots, \mathbb{F}_{\Phi,x}$ of equal (or almost equal) size. One of these sub-sets is now used for testing; the others are used for training the selector, as shown in fig. 6.11. As there are x sub-sets, this procedure can be applied x times, each time with a different sub-set for testing (and using the rest for training). How many sub-sets to be used is a parameter that can be adjusted; setting x to 10 is suggested by practical experience [333, p. 126] and has hence been set as default. Since splitting $\mathbb{F}_\Phi$ into x sub-sets also involves randomness (regarding the order of the elements in $\mathbb{F}_\Phi$), the whole process needs to be repeated several times.

The most extreme case of cross-validation is where $x = |\mathbb{F}_\Phi|$, i.e., each element of $\mathbb{F}_\Phi$ is put into a single sub-set. This is called *leave-one-out cross-validation* [333, p. 127] and is also supported by the SPDM plug-in. While this is computationally expensive — x selectors need to be generated, each learning from $x - 1$ instances — it also has some interesting advantages. The splitting of $\mathbb{F}_\Phi$ is not random anymore, hence repeating the whole procedure is unnecessary. Leave-one-out cross-validation can also be regarded as a general epistemological approach to test a hypothesis, as Forster argues in [84].

Selector Performance Measures

The importance of a suitable performance metric for selectors is undisputed, yet it may spur controversy even when it comes to the choice of machine learning method: while Leyton-Brown et al. dismiss classification methods because their error metrics do not distinguish between different kinds of misclassification [205, p. 36–37], Guo and Hsu maintain that classifiers *"also penalize misclassifications differently if being used properly"* [117, p. 66].

Anyhow, all authors agree that errors should not be treated as booleans: a selection mapping that always picks the second-best algorithm is to be preferred over one that always picks the worst one, even though both never pick the 'right' algorithm. The kind of penalization is to some extent domain-dependent: while one metric may test whether the selected algorithm is always within a

certain distance to the optimum, e.g., 10%, another one may simply sum up all errors, i.e., the overall overhead. Yu et al. [338] evaluate their selectors by comparing them to an optimal selector, a random selector, and a constant selector that always picks a given algorithm (i.e., an element of $\mathbb{S}_C$, see def. 2.1.6, p. 18). In [96], Gagliolo and Schmidhuber measure the *cumulative overhead*. It represents the relative overhead of their AOTA framework (see sec. 2.5.3, p. 42) with respect to an optimal execution, over a number of trials. Vuduc et al. [317, 318] report the percentage of missed optimal configurations, Δ_{miss}, and the average slow-down of the selected configurations, Δ_{err}. They also consider how often a selector chooses a configuration from the best or worst $X\%$ of the available options. Such metrics presume detailed information on the performance space, which is not always available.

In the SPDM, a performance measure is defined on a test set of performance tuples, Φ_{test}, i.e., tuples that have not been used to generate the selector. Since an SPDM selector — as discussed in section 6.2.1 (p. 119) — merely compares the (predicted) performance of two algorithms $a_1, a_2 \in \mathbb{A}$ when applied to a problem with features $f \in \mathbb{F}$, it can be formally defined as a function $Sel : \mathbb{F} \times \mathbb{A} \times \mathbb{A} \to \{-1, 0, 1\}$. This does not restrict the selector, internally is still may predict each algorithm's performance individually (see eq. 6.1, p. 120, and eq. 2.13, p. 25).

The performance measure returns a tuple $(e_\Sigma, e_n) \in \mathbb{R} \times \mathbb{N}$ that gives the error sum and the number of conducted predictions, which may differ among selector performance measures. The expected error can then be estimated by $\frac{e_\Sigma}{e_n}$. The following paragraphs describe the selector performance measures that have been implemented so far.

Boolean Measure of Mispredictions This error measure implements an overly simplistic metric, against which Leyton-Brown et al. argue in [205]: it just counts all wrong comparisons between two performance tuples that have the same features. It is hence mainly interesting for basic error analysis, but also serves as a foundation for more complicated measures. The boolean error measure of a selector Sel is calculated by:

$$\sum_{f \in \mathbb{F}_{\Phi_{test}}} \left(\sum_{\phi_1, \phi_2 \in \Phi^f_{test} \wedge \phi_1 \neq \phi_2} c(\phi_1, \phi_2) \right) \tag{6.3}$$

with $c : \Phi_{test} \times \Phi_{test} \to \{1, 0\}$ being defined as

$$c((f, a_1, p_1), (f, a_2, p_2)) = \begin{cases} 1 & Sel(f, a_1, a_2) < 0 \wedge p_1 \geq p_2 \\ 1 & Sel(f, a_1, a_2) > 0 \wedge p_1 \leq p_2 \\ 1 & Sel(f, a_1, a_2) \neq 0 \wedge p_1 \neq p_2 \\ 0 & \text{otherwise.} \end{cases} \tag{6.4}$$

Note that the features f refer to *both* performance tuples given to c, which is ensured by the definition of Φ^f_{test}. The function c returns 1 for every misclassification that occurs. In its given form, it is only applicable in case the selection mapping aims at *minimizing* the given metric, e.g., because it is a consumptive algorithm performance measure like execution time (see sec. 5.1.1, p. 108). This can be seen, for example, in the first line on the right-hand side of equation 6.4: if a_1 is preferred over a_2 — i.e., $Sel(f, a_1, a_2)$ returns a negative value — but the performance p_1 of a_1 is *greater* than that of a_2, this is considered as a misclassification. As with the following measures, it is straightforward to adapt equation 6.4 to selectors that shall maximize performance instead.

Numeric Measure of Mispredictions In case the selector Sel predicts the performance of algorithms in isolation, i.e., it follows the approach given in equation 2.13 (p. 25), the simple boolean measure can be extended to *quantify* the extent to which algorithm performance is mispredicted. Formally, is can be calculated by

$$\sum_{\phi \in \Phi_{test}} c'(\phi)$$

with $c' : \Phi_{test} \to \mathbb{R}_0^+$ being defined as

$$c'((f, a_1, p_1)) = \frac{|p_1 - \widehat{perf}_{a_1}^{Sel}(f)|}{\max(\widehat{perf}_{a_1}^{Sel}(f), p_1)}$$

where $\widehat{perf}_{a_1}^{Sel}(f)$ is the performance predictor used by selector Sel for algorithm $a_1 \in \mathbb{A}$ (cf. eq. 2.13, p. 25). Note the normalization by $\max(\widehat{perf}_{a_1}^{Sel}(f), p_1)$, i.e., this measure gives the relative error, summed up over all test tuples.

Misprediction Regret While a quantitative evaluation of mispredictions already gives a good picture of *overall* selector performance, this might still be the wrong metric to be optimized. What matters in the end is the performance difference between the automatically selected algorithm and the *best* one, a notion similar to the regret of reinforcement policies discussed in section 2.3.2 (p. 32).

Formally, this performance measure sums up the relative performance loss due to using the selected instead of the overall best algorithm for every feature $f \in \mathbb{F}_{\Phi_{test}}$:

$$\sum_{f \in \mathbb{F}_{\Phi_{test}}} \left(\frac{Sel_{perf}(f)}{\min\{p | (f, a, p) \in \Phi_{test}^f\}} - 1 \right) \tag{6.5}$$

where $Sel_{perf}(f)$ denotes the performance of the algorithm selected by Sel when confronted with features f. Note that, again, the best performance is defined here to be the minimal performance. Subtracting 1 just leaves the relative *overhead*, i.e., the speed-up of using the (unknown) best selection mapping S^* (see def. 2.1.2, p. 15) instead of the learned selection mapping represented by Sel.

Furthermore, it is important to notice that the above definition implicitly assumes that the performance data only contains problems with unique features. Otherwise, the regret calculation in 6.5 becomes ambiguous: imagine two problems $x_1, x_2 \in P$ with the same features $f \in \mathbb{F}$ have been used to generate Φ. For any algorithm $a \in \mathbb{A}$ that Sel selects, there will be *two* performance tuples $(f, a, p_1), (f, a, p_2) \in \Phi_{test}$ — which of them is $Sel_{perf}(f)$ in the nominator of equation 6.5? This constraint is automatically checked when the misprediction regret shall be computed.

6.2.3. Additional Components and Overview

The realization of the two main purposes of SPDM, selector generation and evaluation, does not suffice to provide a fully automated data analysis. It is equally important to link all the exchangeable components — selector generators, evaluation strategies, and selector performance measures — in a meaningful way that allows their easy application to the data stored in the performance database. This is implemented by another central component, `SelectorGeneratorEvaluation`, a simple class that provides a lean interface to the underlying functionality. It allows to configure which selectors should be generated from which performance data, and how these should be evaluated. Then, it carries out all necessary operations and returns the best selector, i.e., it implements a pragmatic approach to induction as described in section 1.4 (p. 8). Figure 6.12 shows the overall structure of the SPDM. The following sub-sections briefly discuss some other relevant components shown in figure 6.12, e.g., for creating a performance data set Φ from some source of performance data.

Data Import

To keep the SPDM framework independent of the source from which performance data is collected, an additional JAMES II plug-in type for data import is defined. Currently, only two alternative

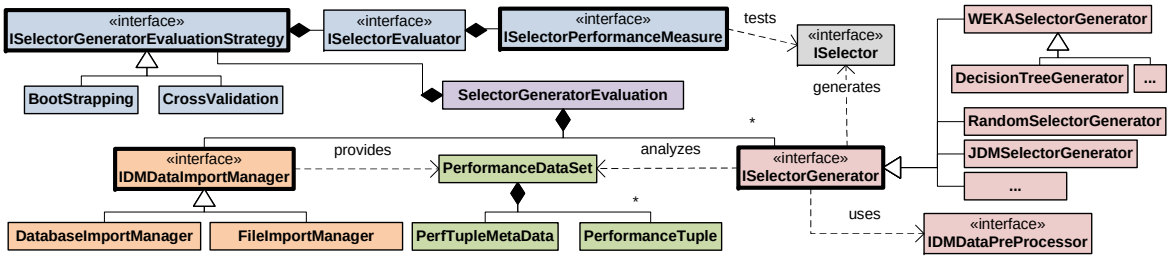

Figure 6.12.: UML class diagram of central SPDM components: data import (orange), representation of performance tuple sets (green), selector generation (red), selector performance measurement and evaluation (blue), and finally a central entity for executing the whole process (purple). The product to be ultimately delivered, i.e., instances of `ISelector`, is shown grey. Interfaces marked with bold borders denote custom plug-in types.

plug-ins are provided: an XML import plug-in allows to read formerly generated performance tuple sets that have been persisted with Java's `XMLEncoder` utility, and an additional plug-in that allows to import data from the performance database (see ch. 5, p. 101).

The database import plug-in encodes all relevant entities from the data base into a performance tuple set Φ, which is complemented by additional meta-data (see sec. 6.2.1, p. 118). It is configured to only extract performance tuples for models with a URI that matches a user-defined string. This allows to restrict the analysis to certain modeling formalisms or specific models. Similarly, only a single type of performance measure (see sec. 5.1.1, p. 108) is considered, which is also set by the user. All simulation problems that have not been processed sufficiently often by all runtime configurations are filtered out, to ensure that learning is done on a basis of well-explored problems only. Users can configure how many applications are required to make the performance data on a simulation problem eligible for import.

Preprocessing

An additional step for preprocessing performance tuples has been laid out, as this might be an important step before selector generation. Furthermore, such mechanisms could realize methods for feature selection, an important adjacent problem already discussed in section 2.1.1 (p. 14). Since the effectiveness of preprocessing components is likely to depend on the selector generation method, a preprocessor is defined by the selector generator factory at hand. The factory returns an instance of `IDMDataPreProcessor`, which is then applied to Φ. Preprocessors may be represented by an additional plug-in type later on, in case their are widely applied.

Further Selector Generators

There are additional selector generators that have not been discussed in section 6.2.1 (p. 118), e.g., the `RandomSelectorGenerator` (see fig. 6.12). It is necessary for performance evaluation to construct 'baseline' selection mappings that realize a random selection, i.e., selectors that exhibit average-case performance in the long run. Only selectors that *outperform* random selection can be regarded as average-effective in the sense of definition 2.1.5 (p. 17): the average performance of their selections has to be higher than the average algorithm performance in general, which is approximated by the random selector. Any selector that fails to outperform random selection hence needs to be discarded.

Following the argumentation in section 2.1.2 (p. 16), average-effectiveness does not imply superiority over a very simple scheme that always sticks with the generally 'best' algorithm. Hence, there also is a selector generator that adopts a *winner-takes-all* philosophy: it generates a selector that always picks the configuration with best average performance, regardless of the problem features. In the nomenclature of section 2.1.2, the generator hence creates a selector that implements a constant selection mapping (from $\mathbb{S}_C$) that performs best on the considered problem set: only a selector that

performs *better* than the *winner-takes-all (WTA)* selector can be regarded as adaptive-efficient (see def. 2.1.6, p. 18).[8]

Finally, a generic `EnsembleSelectorGenerator` has been developed. It uses another selector generator to create an individual selector for every single algorithm — i.e., configuration — encountered in Φ. Decisions are made by querying each individual predictor and then comparing their predictions. Since the performance classes distinguished by individually trained classifiers do usually not match, i.e.,'good' means something different to each of them, the ensemble method is currently restricted to regression methods, i.e., methods that deliver quantitative performance predictions. Such a scheme is further restricted to situations where the size of $\mathbb{A}$ is sufficiently small, yet it may reduce the complexity of distinguishing between different algorithms in a single mapping. Similar schemes would also allow the application of meta-learning (see sec. 2.7, p. 55).

6.2.4. Current Limitations

While the current solution allows to derive selection mappings from performance data in a simple manner, it should also be clear what SPDM does *not* deliver (yet): it does neither offer techniques for feature selection (see def. 2.1.3, p. 15), nor does it support any kind of interactive performance analysis.

Feature selection may improve the quality of the selectors produced by SPDM, e.g., the size of a decision tree, and may also speed up selector generation (because many features might be dismissed). A thorough evaluation of the potential impact of feature selection has been carried out by Leyton-Brown et al. in [205]: they were able to reduce the number of features from 30 to just 5–8 while preserving the quality of the selection mappings. Moreover, they show how to consider the trade-off between features that are expensive to compute and the potential performance gains from using them for better algorithm selection. A simple algorithm that builds additional statistical models to predict the precision and performance gains is introduced to do so [205, p. 39]. Besides exploiting the correlations between features to identify the most useful ones, the same knowledge can be applied by performance analysts to construct particularly hard benchmark instances [205, p. 39–42].

While interactive performance analysis is an important tool for analysts and developers alike, it is — unlike feature selection — nothing that is required to fulfill the main SPDM task, i.e., automated selector generation and evaluation; rather, a dedicated tool that *employs* the SPDM for performance data mining tasks should be developed. Nevertheless, a graphical representation of selection mappings and performance data facilitates comprehension [184]. Vuduc et al. use *truth maps* [318], i.e., two-dimensional plots with a point cloud where each item says which algorithm performed best on this particular problem instance.[9] Clearly, this technique does not scale with the number of problem features of interest and it does not show the performance difference to the runner-up. Other visualizations show prediction error (e.g., [319]), compare the regions where the performance relation between algorithms changes (or is predicted to do so, e.g., [25]), or provide approximated cumulative distribution functions of execution time (e.g., [114]).

6.3. Summary

This chapter described the layout and implementation of the SPDM framework for simulator performance data mining. The SPDM allows to integrate external data mining tools; wrappers for Weka, Joone, and MLJ have already been realized (see sec. 6.2.1, p. 121). All tools are accessed through a common interface. Their results are evaluated by re-implementations of established methods from data mining, in combination with custom performance measures (see sec. 6.2.2, p. 123). Data import from a custom XML format and also from the performance database (ch. 5, p. 101) is supported. Additional components have been added to account for the theoretical considerations of section 2.1.2

[8] Note that this *only* applies to the problem set that is considered for training: a WTA selector does not necessarily implement the best-performing constant selection mapping for other problem sets.

[9] Similar diagrams are used in [25].

(p. 16), so that average-effectiveness and adaptive-effectiveness of a selector can be checked and the best selector can be identified automatically.

While users of all roles may access the SPDM similarly — i.e., via selector generator evaluation — their role is likely to affect the choice of performance measurement: while deployers are interested in the difference to an optimal selection and need to compare selectors to the winner-takes-all and random selection strategies, performance analysts might be more interested in the predictability of the algorithm performance as such, e.g., the numerical measures for misprediction (see sec. 6.2.2, p. 125). The SPDM does not yet cover feature selection techniques — but this could be added to the overall process later on, without having to change the interface with which deployers and performance analysts operate.

It is now clear how performance data is measured and stored (ch. 5), and also how the SPDM generates *selectors* from this data that represent selection mappings. Another important step of the overall performance analysis process is still missing: how to set up experiments that allow to collect such large amounts of performance data? As it turns out, this takes several precautions and additional components. These are discussed in the next chapter, which focuses on performance experiments with JAMES II. Interestingly, another ASP solution mechanism can be integrated within this process, which does not necessarily require the analysis of past performance data.

7. Experimentation Methodology

You are invited to come to see the Earth turn, tomorrow, from three to five, at Meridian Hall of the Paris Observatory.

Invitation card by Jean Foucault, 1851

The general challenges of experimenting with simulation algorithms have already been discussed in chapter 3 (p. 61). This chapter builds up on that by briefly outlining how these challenges are currently met by the experimentation layer of JAMES II (sec. 7.1). Section 7.2 introduces a simple yet powerful algorithm selection technique that can be plugged into the experimentation layer. It both explores and exploits algorithm performance at the processing time of a simulation problem, but without relying on the performance database or the data analysis methods presented in chapter 5 (p. 101) and chapter 6 (p. 115). Then, section 7.3 shows how combining this technique with additional components facilitates the execution of large-scale experiments to investigate the runtime performance of simulation algorithms. A calibration method that yields significant speed-up for such experiments is also introduced (sec. 7.3.2). Finally, it is illustrated how all these techniques can be used in conjunction with the performance database, to automatically set up meaningful performance experiments for developers (sec. 7.3.4). All in all, three elements of the SASF are covered in the following (see fig. 7.1). Many related algorithm selection approaches also rely on sophisticated setups for performance experiments; these are discussed as related work in the corresponding sections.

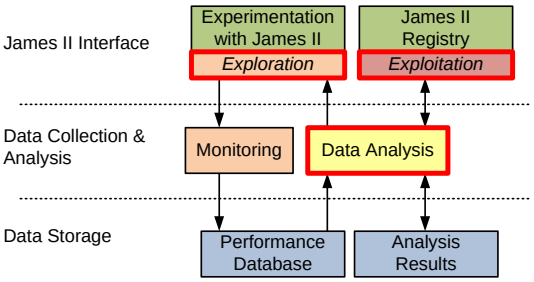

Figure 7.1.: SASF overview (see fig. 4.6, p. 97), red borders denote elements discussed in this chapter.

7.1. The Experimentation Layer of JAMES II

The development of an experimentation layer for JAMES II has been a group effort right from the beginning [68, 74, 132, 201]. It is a complex and feature-rich part of JAMES II that cannot (and need not) be described here in full detail; the following discussion is restricted to a brief overview of the main concepts and then centers on those entities that are exploited or extended later on.

The major design goals of the experimentation layer have already been motivated in section 3.1 (p. 61): reproducibility and comparability (e.g., [101, 163]). Some issues — e.g., random number generation [74] — are resolved by dedicated sub-systems, while others — e.g., flexibility regarding experiment design techniques (see sec. 3.2, p. 66) — require structural precautions.

Providing users with such a layer on top of the JAMES II plug-in system (see sec. 4.2, p. 81) allows to leverage the strengths of a plug-in based approach for conducting scientifically sound simulation studies. The desire of users to have a flexible and thorough support for experimentation is not new (e.g., [137]) and has been implemented in many other simulation systems as well, e.g., Tornado, where a-priori experiments are used for (manual) algorithm selection [41, p. 145 et sqq.]. Few simulation systems, however, are able to integrate a comparable range of techniques from different fields, let alone allowing users to combine them flexibly.

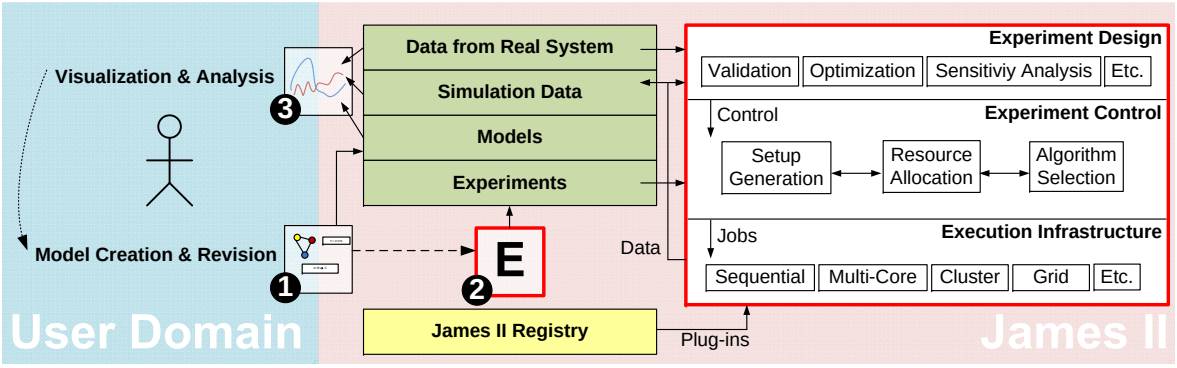

Figure 7.2.: The experimentation layer of JAMES II (from [68]): After modeling (step 1), setting up simulation experiments (step 2) is necessary before any output can be analyzed (step 3). Arrows denote either data or control flow. While the experimentation layer (red box, right) comprises various components, the discussion here will be restricted to experiment design and control, i.e., some elements from the upper two layers of the red box.

The JAMES II experimentation layer relies on distinct components for defining experiments and for controlling their execution. Figure 7.2 shows an overview of the layer and its relation to other JAMES II components.

Experiment Definition

JAMES II experiments are defined by instances of the `BaseExperiment` class. It contains all information required to execute an experiment, e.g., from where to load the model, what data to observe, where to store the observed data, and so on. Since all these tasks are carried out by JAMES II plug-ins, a `BaseExperiment` merely stores the factories (and their parameters). Most importantly, a `BaseExperiment` also contains an instance of `ExperimentVariables`, a hierarchical structure that defines the model parameterizations to be evaluated. To conduct a simple parameter scan, it is sufficient to add so-called *experiment variables*, i.e., instances of the class `ExperimentVariable`, to an `ExperimentVariables` instance. The latter contains a list of experiment variables and a reference to another instance of `ExperimentVariables`, which represents a lower level of the hierarchy.

The default implementation of `ExperimentVariables` changes the values of *all* experiment variables on its own level *simultaneously*. However, this is only done if its sub-list — i.e.,, the `ExperimentVariables` instance representing the hierarchy below — signals that all parameter combinations defined by the lower lists have already been processed. Otherwise, the variable values stay the same on the given level. In case the values of the variables cannot be changed any further, this is signaled to the parent list. These simple rules allow to easily define different factorial experiments, ranging from a full factorial, i.e., each variable is in its own list, to setups where all variable values shall be changed at once, i.e., all variables are defined in a single list. This is illustrated in figure 7.3 (p. 133).

Subclasses of `ExperimentVariables` allow the integration of experiment design techniques (see sec. 3.2, p. 66) or optimization algorithms, i.e., any non-trivial method to choose the model parameter values of interest. Since these methods *steer* the experiment into certain parameter regions, they are subsumed under the term *experiment steerers*.

Techniques like optimization also require that observed data is fed back to them, in order to calculate the objective value attained by a specific parameterization (see sec. 2.1.3, p. 20). The feedback propagation is realized by the experimentation layer as well. Note that instances of `ExperimentVariables` subclasses may still be combined with instances of the original class; this allows, for example, to evaluate the performance of various optimization algorithms when applied to to the same simulation-based optimization problem. However, not all kinds of nesting are meaningful. For example, adding addi-

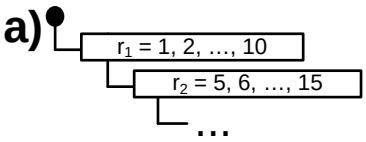

Figure 7.3.: The default implementation of experiment variables (from [68]): the hierarchy defined in setup *a)* consists of two `ExperimentVariables` instances, each defining a single experiment variable (r_1 and r_2) so that all value combinations are generated (i.e., full factorial): $(1,5),(1,6),\ldots(10,15)$. In contrast, setup *b)* merely defines 10 setups: $(1,5),(2,6),\ldots,(10,15)$.

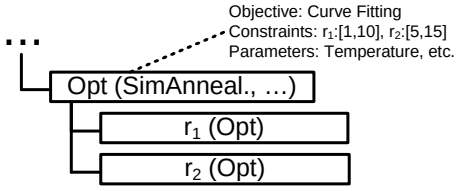

Figure 7.4.: Extending JAMES II experiment variables for optimization: no additional sub-lists are allowed. The optimization algorithm (here, simulated annealing) assigns new values to r_1 and r_2 that are not known before runtime.

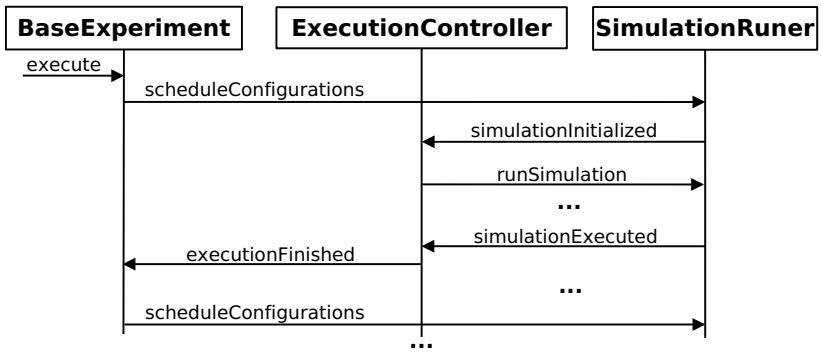

Figure 7.5.: Interplay between `BaseExperiment`, experiment execution controller, and simulation runner.

tional levels containing experiment variables *below* an optimization algorithm will make it consider whole *sets* of model setups for each parameter combination under its control. Such setups are therefore not supported. The integration of experiment steerers, exemplified by an optimization algorithm, is depicted in figure 7.4 (p. 133).

To accommodate methods discussed later in this chapter, experiment variables need to control not only model parameters, but also execution specifics, e.g., which algorithms to use, when to stop the simulation, and so on. This is realized by a `DynamicExperimentConfigurator`, which collects changes to the current configuration from all experiment steerers defined for the given experiment. Since a runtime configuration is expressed in form of nested parameter blocks (see fig. 6.5, p. 119), additional components have been developed to facilitate their manipulation (package `james.core.experiment.execonfig`). They allow to define rules that apply to certain subsets of nested parameter blocks. This mechanism can be used, for example, to set event queue parameters in *all* relevant parameter blocks — the queue might be chosen more than once — or to more easily update a certain algorithm parameter that is nested deeply within the parameter blocks.

Experiment Control

Upon execution, the `BaseExperiment` instance generates a certain number of so-called *simulation configurations*. A simulation configuration can be regarded as a job description that contains all information to execute a certain model setup with a certain configuration of the simulation system. A simulation configuration also contains *replication criteria*: plug-ins that decide how often a simulation problem should be replicated, e.g., until a certain level of statistical significance has been reached.

As the sequence diagram in figure 7.5 shows, the `BaseExperiment` instance sends the configurations to a so-called *simulation runner*, which is responsible for executing it. The execution as such is

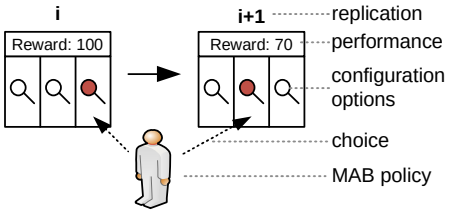

Figure 7.6.: Adaptive replication: mapping the multi-armed bandit problem onto algorithm selection during simulation replication (cf. fig. 2.7, p. 33).

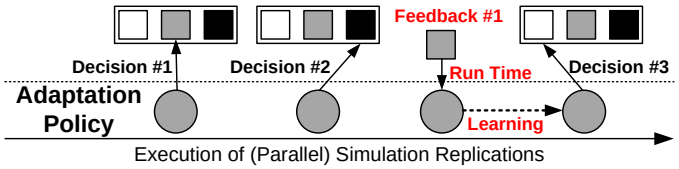

Figure 7.7.: Asynchrony of reward reception: the policy has to select algorithms (black, white, and grey box) without knowing all past rewards.

transparent to the experiment that generates the simulation configurations, which makes it easy to support different execution strategies for different kinds of infrastructure, as discussed in [201]. Again, the flexibility of choosing different simulation runners is realized by defining a corresponding plug-in type and plug-ins.

An additional component, the *experiment execution controller*, mediates status updates between `BaseExperiment`, simulation runner, and user interface. The latter may require to withhold the execution of a certain simulation run, e.g., because the user first needs to configure an online visualization. Hence, the simulation runner notifies the controller when a simulation run is ready to be executed, i.e., it has been properly initialized. In turn, the controller triggers the simulation runner to execute (or cancel) the job. In case the execution finished, a notification is sent from the simulation runner to the execution controller, from where it is propagated to the `BaseExperiment`. After the scheduled set of simulation configurations has been processed, the `BaseExperiment` may either schedule another set of jobs or stop the execution because it is finished.

7.2. An Adaptive Simulation Runner

Simulation runners have a simple interface and separate the concerns of simulation execution and experiment execution, the latter being realized by `BaseExperiment`. While the execution of a simulation run as such is clearly the responsibility of the simulators and their sub-components (see [131] for details), the *scheduling* of the simulation runs to the available resources, i.e., processors, is what the simulation runner has to implement.

As it directly controls the execution, a simulation runner can easily observe (or be informed on) any consumptive performance measure of the current run (e.g., execution time or memory requirements; see sec. 5.1.1, p. 108). In case *multiple* replications have to be conducted, i.e., the same simulation problem has to be solved multiple times, this feedback can be used to select algorithms: the whole situation is very similar to the multi-armed bandit problem discussed in section 2.3.2 (p. 32). Multiple replications are almost always required for stochastic simulation — and stochastic simulation approaches and models are abundant, e.g., to simulate chemical reaction networks (see sec. 1.3.1, p. 4). A suitable algorithm selection approach on this level may already alleviate the algorithm selection problem in many cases.

Figure 7.6 illustrates the basic idea of *adaptive replication*: the set $\mathbb{A}$ of eligible algorithms is now represented by the arms of the bandit, as already discussed in section 2.3.2 (p. 32). Each simulation run to be executed, i.e., each replication, requires to select an element from $\mathbb{A}$, i.e., to choose an arm. The performance of the selected algorithm when applied to execute a single replication can be interpreted as reward. In contrast to the original multi-armed bandit problem, several restrictions and requirements arise in the context of simulation replication:

- *Minimization*: Since the policies typically select algorithms based on *consumptive* algorithm performance measures (see sec. 5.1.1, p. 108), they have to minimize the overall reward instead

of maximizing it. This is reflected by some straightforward adjustments to the policies, as discussed in section 7.2.1.

- *Manual restriction of options*: The restriction to consumptive performance measures — or, more technically, all performance measures that can be observed from a *single* replication — makes it necessary to allow the specification of the algorithm subset $A \subset \mathbb{A}$ that shall be explored. This enables users (or external mechanisms) to pre-select those methods that fulfill all requirements regarding more elaborate performance metrics, e.g., concerning the accuracy (see sec. 5.1.1, p. 108).

- *Asynchrony*: Simulation runners may exploit multiple resources and schedule simulation executions to run concurrently, i.e., they may conduct parallel independent replications [201]. Hence, a policy may have to decide which arm to pick *before* it received all — or, in fact, *any* — reward, as illustrated in figure 7.7. In contrast, the multi-armed bandit problem assumes that policies have access to *all* rewards their prior decisions brought about.

- *Faulty configurations*: Apart from being more or less suitable to solve a given simulation problem, a runtime configuration may also fail to work at all, i.e., it stops with an error message.[1] The error message is detected by JAMES II and gets propagated to the simulation runner. A policy should differentiate between a bad reward and a faulty configuration: while the configuration yielding little reward might eventually be reevaluated, a faulty configuration should not be chosen anymore. In other words, the policy should *quarantine* the configuration for the given simulation problem.

The implementation of an *adaptive simulation runner* has to take into account all of the above issues. An adaptive simulation runner should also be flexible with respect to the specific policy that is employed, since it is unclear which one will perform best. The convergence speed of a policy, i.e., how many replications it takes until the best algorithm is identified, is problem-dependent. It relies on the shape of the reward distributions (e.g., see the runtime distributions of simulation algorithms, fig. 2.11, p. 46). Section 7.2.1 details the implementation of the adaptive simulation runner [72] and some policies, while section 7.2.2 introduces an approach to improve policy convergence speed [75].

Related Work

An alternative approach to speed up the execution of multiple replications has been developed for parallel and distributed discrete-event simulation. The basic idea of *cloning* [152, 39] a parallel simulation is to compute multiple trajectories of a model in a single execution, so that replications can share common calculations. The simulation starts out with a single trajectory and proceeds as usual, until a certain *decision point* is reached. Then, for each potential outcome at the decision point, the LPs that depend on the outcomes are *cloned*, i.e., they are recreated with the same state and now compute the model trajectory based on this outcome, while their counterparts compute trajectories for other outcomes. This is particularly useful if only small and isolated parts of the model are affected by the outcomes of a decision point, so that much computation can be shared among all replications. Unfortunately, this is not always the case; e.g., stochastic models in systems biology usually do not exhibit this behavior.

Another approach from the domain of PDES, *active replication*, improves the speed of a single execution by letting several algorithms run in parallel per processor, each time using the result of the fastest one while dismissing those of the others [267, 268]. Multiple replications of PDES runs may also be executed on the same resources in parallel, which achieves additional speedups [23].

[1] Clearly, only *certain kinds* of errors result in an error message and can hence be detected by JAMES II, or, for that matter, *any* software system at all. A well-known counterexample is the halting problem, which is undecidable (see sec. 2.1.3, p. 20).

7.2.1. Implementation

General Structure

The adaptive simulation runner reuses the functionality of the parallel simulation runner presented in [201]. A new plug-in type for multi-armed bandit policies that minimize reward is introduced, it subsumes implementations of the interface `IMinBanditPolicy`. The interface defines a method to initialize the policy with the number of available arms and the horizon (see sec. 2.3.2, p. 32), i.e., the number of replications to be expected. The actual horizon is hard to determine, as replication criteria may base their decision on simulation outcomes, i.e., the horizon gets recalculated at runtime. The interface also comprises methods to quarantine algorithms, to request the next decision, and to pass received rewards to the policy. An algorithm is quarantined for a given problem after its execution failed. The next decision is requested by the adaptive simulation runner in case a new simulation run execution shall be scheduled, e.g., because a resource is idle. Similarly, rewards are propagated to the policy when the runner gets notified upon the completion of a run.

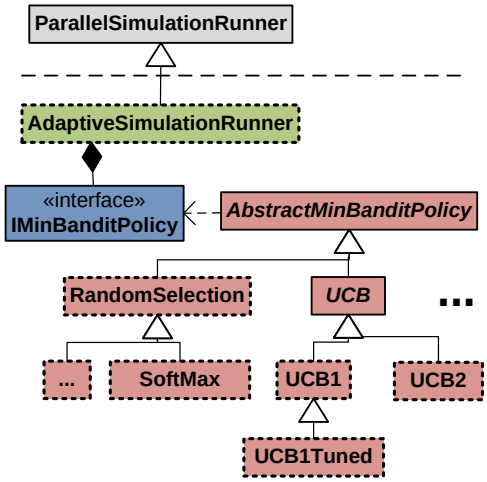

Figure 7.8.: UML class diagram of the adaptive simulation runner and auxiliary components (from [72]): existing components (grey), the simulation runner implementation (green), the policy interface (blue), and the policy implementations (red). JAMES II plug-ins are denoted by dashed borders.

Figure 7.8 summarizes the basic class structure of the adaptive simulation runner. An additional component, `SelectionTreeSet`, is used to automatically generate all suitable plug-in combinations for a given simulation problem $x \in \mathbb{P}$, i.e., the whole set $\mathbb{A}_x$ of eligible selection trees (see eq. 5.1, p. 105). To construct $\mathbb{A}_x$, `SelectionTreeSet` considers the given simulation problem and analyzes the plug-in data from the JAMES II registry (see sec. 4.2.1, p. 84). Plug-in descriptions specify on which components (of which plug-in type) a plug-in relies. This information can be used to create all eligible selection trees. Note that `SelectionTreeSet` does not distinguish between variations in algorithm parameters, i.e., an automatically generated set $\mathbb{A}_x$ only contains plug-in combinations with default parameters. Anyhow, this allows to use the adaptive simulation runner without any user interference. Alternatively, a set of pre-selected runtime configurations can be defined, represented by corresponding parameter blocks.

Multi-Armed Bandit Policies for Replication

As all multi-armed bandit policies have to manage quarantined arms and other common parameters (e.g., horizon, number of arms), they are implemented as subclasses of the class `AbstractMinBanditPolicy` (see fig. 7.8), which implements their commonalities. The following para-

graphs explain some of the policies that have been implemented so far, and how they had to be changed to suit the context. For example, the asynchrony with which rewards are received (see fig. 7.7) makes selection difficult for policies that simply do not cover this case, e.g., because they can only make a decision after trying out each arm once.

Fortunately, such problems can often be resolved by relatively small adjustments. If not stated otherwise, the problem of asynchronous rewards is solved by applying a simple heuristic: in case no reward has been received yet, a policy either draws arms randomly or round-robin (depending on whether it is a stochastic policy or not). After the first reward has been received, the corresponding arm is regarded to have the minimal reward so far, i.e., it is deemed the best one; all arms without known reward are excluded from exploitation. The rule of choosing the best arm from those where reward has already been received lends itself particularly well to the execution time performance measure: it is likely that the execution of faster algorithms finishes earlier than the execution of slower algorithms, given equal external load. Hence, the algorithms that are selected before the required number of rewards have been received are likely to be good choices anyway. Many basic approaches to solve the multi-armed bandit problem are realized by various slightly different policies. The following paragraphs are organized around such 'families' of policies. Since rewards shall be minimized and not maximized, the existing policies had to be adapted and will be described accordingly.

ϵ Policies This family of policies is controlled by a single parameter, ϵ, that governs the trade-off between exploration and exploitation (see sec. 2.3.2, p. 29). The most simple policy, ϵ-First [316], plays the first $\lceil \epsilon \cdot n \rceil$ rounds by selecting arms randomly, n being the horizon. Then, the arm with minimal average reward $\hat{R}_i$ is used for the rest of the rounds. ϵ-Greedy [316] does not stop the exploration after a predefined number of rounds, as ϵ-First does. Instead, it randomly chooses each round either to explore a random arm (probability ϵ) or to play the best arm (probability $1-\epsilon$). While the exploration phase of ϵ-First may simply be too short and too undirected, ϵ-Greedy carries out an undirected exploration over the whole horizon. This, in turn, may lead to too much exploration, since the probability ϵ is constant. Even after much knowledge has been gained, the policy still explores alternatives with the same effort.

To avoid this, the ϵ-Decreasing [316] policy explores a random arm with a probability of $\frac{\epsilon}{r}$, where r is the number of rounds that have been played so far.[2] An alternative variant is called ϵ-GreedyMix in [316] and calculates the exploration probability as $\frac{\epsilon \cdot \log_{10} r + 1}{r}$.[3] Yet another variant of ϵ-Decreasing — ϵ_n-Decreasing — is proposed by Auer et al. and defines the exploration probability as follows [11, p. 239]:

$$\epsilon_n = \min\left(1, \frac{c \cdot k}{d^2 \cdot r}\right)$$

where r is again the number of rounds that have been played, k is the number of arms, and $c > 0$ as well as $d \in (0, 1)$ are parameters. The parameters control the policy's propensity for exploitation and exploration: if c is increased, the probability of choosing a random arm is increased, while d increases the probability of exploitation by choosing the best arm so far. ϵ_n-Decreasing is particularly interesting because Auer et al. could derive an upper bound on its regret (see sec. 2.3.2, p. 32), i.e., the performance loss due to exploration. However, their proof requires some additional assumptions on the reward distributions [11, p. 240].

Finally, ϵ-LeastTaken [316] is very similar to ϵ-Decreasing in that it also lets the probability of exploration decrease over time, depending on ϵ, but now the exploration is directed: the exploration probability is defined as $\frac{4 \cdot \epsilon}{4 + m^2}$, where m is the number of times that the arm that has been selected the *least* was tried. If the ϵ-LeastTaken policy decides for exploration, it will consequently explore this

[2] Note that in the beginning, as mentioned earlier, a random arm is chosen: $r \geq 1$.

[3] The original formula given in [316] is $\frac{\epsilon \cdot \log_{10} r}{r}$ — however, this would mean that the exploration probability during the first 10 rounds is *less* than ϵ. The addition of 1 in the dividend overcomes this problem and does not affect policy performance in the long run, as $\frac{1}{r}$ approaches zero for $r \to \infty$.

least selected arm, i.e., the arm for which the reward distribution is approximated from the smallest sample of observed rewards.

UCB Policies The policies in this family are deterministic; they do not involve any random choice. The basic idea behind them is to calculate an *upper confidence bound* (UCB) for each arm's reward. This confidence bound is very optimistic at first, due to the lack of data. As a policy receives more and more rewards from choosing a certain arm, its confidence bound will be adjusted and eventually gives a realistic picture of the reward to be expected. One could regard all UCB policies as greedy heuristics, as they *always* play the arm with the maximal index, i.e., the arm with the maximal upper confidence bound. Overall reward should be *minimized* here, so the actual UCB implementations choose the arm with the *least* index instead, and subtract estimates on the confidence bounds accordingly. In other words, they consider a *lower* confidence bound on rewards. UCB policies mainly differ in their calculation of arm indices. All policies discussed here have shown to be zero-regret strategies, i.e., they are guaranteed to eventually identify the best arm and play it exponentially more often than any other [11].

UCB1 [11] calculates the index of arm i as

$$\hat{R}_i - \sqrt{\frac{2 \cdot \ln r}{r_i}} \tag{7.1}$$

where $\hat{R}_i$ is the average reward received from pulling arm i, r_i is the number of times arm i has been selected so far, and r is again the number of rounds played. For UCB1-Tuned [11], this formula is changed to

$$\hat{R}_i - \sqrt{\frac{\ln r}{r_i} \min\left(\frac{1}{4}, V_i(r_i)\right)} \tag{7.2}$$

where $V_i(r_i)$ estimates the variance (bounded by $\frac{1}{4}$, see [11, p. 245]) of the arm's rewards:

$$V_i(r_i) = \left(\frac{1}{r_i}\sum_{j=1}^{r_i} R_{i,j}^2\right) - \hat{R}_i^2 + \sqrt{\frac{2 \cdot \ln r}{r_i}} \tag{7.3}$$

where $R_{i,j}$ is the reward received by selecting the i-th arm for the j-th time. Finally, UCB2 [11] calculates the index of arm i as

$$\hat{R}_i - \sqrt{\frac{(1+\alpha) \cdot \ln \frac{e \cdot r}{\tau(e_i)}}{2 \cdot \tau(e_i)}} \tag{7.4}$$

where $\tau(x) = \lceil (1+\alpha)^x \rceil$, $\alpha \in (0,1)$ being a parameter, and e_i being the number of epochs for which arm i was chosen. An epoch represents a number of rounds during which the policy is committed to play the selected arm. Each epoch has $\tau(e_i+1) - \tau(e_i)$ rounds, i.e., its length depends on the number of epochs arm i has already been played, and also on the parameter α. If α is increased, the length of the epochs will grow faster.

As all UCB policies require a first estimate $\hat{R}_i$ for the average reward, they first choose each of the available arms once. This initialization sets a natural upper limit for the number of arms that is supported: if it exceeds the number of replications, no performance gain by automated algorithm selection is possible. UCB policies also assume reward distributions with support in $[0,1]$. This is currently handled by a normalization (and index recalculation) whenever a new maximal reward is received, which makes it necessary to keep book of past rewards.

IntEstim The interval estimation policy IntEstim [172, p. 51 et sqq.], as adjusted by Vermorel and Mohri [316, p. 5] to cope with rewards from $\mathbb{R}$, follows a strategy similar to the UCB policies: after initially selecting each arm once, mean and standard deviation are estimated to calculate an arm's index. A parameter $\alpha \in (0, 1)$ allows to adjust the confidence bounds, to make the policy more or less aggressive. A variant of IntEstim — IntEstimDec — gradually decreases the impact of an arm's performance variance on the decision by adjusting the parameter α, similar to ϵ-Decreasing for the ϵ policies. Again, it discourages exploration if there is sufficient knowledge to focus on exploitation.

Pursuit The Pursuit policy [298, p. 43] is similar to ϵ-Decreasing in that it chooses arms randomly. At first all arms $1, \ldots, k$ are selected with the same probability $pr_i = \frac{1}{k}$. The selection probabilities are adjusted after receiving a reward. The probability of the arm with the lowest $\hat{R}_i$ — i.e., the best arm — is increased: $pr_{best} = pr_{best} + \beta \cdot (1 - pr_{best})$. At the same time, the probabilities of all other arms are decreased: $pr_{i \neq best} = pr_i - \beta \cdot pr_i$. These manipulations do not change the sum of the probabilities. If arm 1 performed best so far (w.l.o.g.), its probability increment of $\beta \cdot (1 - pr_1)$ is compensated by decreasing all other probabilities:

$$\beta \cdot (1 - pr_1) - \beta \cdot pr_2 \ldots - \beta \cdot pr_k = \beta - \beta \cdot pr_1 - \beta \cdot pr_2 \ldots - \beta \cdot pr_k = \beta - \beta(pr_1 + \ldots + pr_k) = 0$$

since the probabilities sum to 1, i.e., $\sum_{i=1}^{k} pr_i = 1$. The parameter $\beta \in (0, 1)$ controls the convergence speed of the policy, i.e., how fast it gives up exploration for exploitation. The order of the $\hat{R}_i$ may change during execution, so arms with non-minimal $\hat{R}_i$ can be thought of pursuing the arm that is currently deemed best (hence the name).

SoftMax Similar to Pursuit, SoftMax [316, p. 4] also selects arms randomly while adjusting the selection probabilities based on the received rewards.[4] SoftMax calculates the probability of selecting arm i as:

$$pr_i = \frac{\exp(\frac{\hat{R}_i^{-1}}{\tau})}{\sum_{j=1}^{k} \exp(\frac{\hat{R}_j^{-1}}{\tau})} \tag{7.5}$$

where the temperature parameter $\tau \in \mathbb{R}^+$ controls the degree of exploration (the higher the more). The reciprocals $\hat{R}_j^{-1} = \frac{1}{\hat{R}_j}$ are used to adapt SoftMax to the minimization requirement.

As with the ϵ policies (e.g., ϵ_n-Decreasing), it seems natural to let τ decrease over time, i.e., to reduce exploration when sufficient observations have been made. A τ-decreasing SoftMax policy — SoftMaxDecr — uses a parameterizable initial temperature τ_0 that is set to $\frac{\tau_0}{r}$ at round r (see [316]). The implementation for the adaptive simulation runner divides τ_0 by the number of rounds for which the rewards has already been received. Another variant — SoftMix — is similar to SoftMaxDecr but sets the temperature to $\tau = \frac{\tau_0 \cdot \log(r)+1}{r}$ (e.g., [316]),[5] where the implementation again only counts the number of rounds for which rewards have already been received.

Reward Comparison RewardComparison [298, p. 41] is rather similar to SoftMax. It employs a SoftMax-like function to calculate the selection probabilities pr_i (cf. eq. 7.5), but replaces the expressions $\frac{\hat{R}_i^{-1}}{\tau}$ and $\frac{\hat{R}_j^{-1}}{\tau}$ with preference values pf_i and pf_j, respectively. When a reward R is received for an arm i, its preference value pf_i is set to $pf_i + \beta \cdot (\bar{R} - R)$, where $\bar{R}$ is the reference reward level used for comparison. If the reward received for the chosen arm compares favorably to $\bar{R}$, i.e., it is *smaller*, the preference value of arm i is increased; otherwise it is decreased. The parameter

[4] Such strategies are called *probability matching policies*, since they aim to approximate the probability of being optimal for each arm.

[5] For the same reasons as given in the description of ϵ-GreedyMix (p. 137), a 1 is added to the dividend. This differs from the formulation in [316].

$\beta \in \mathbb{R}^+$ controls the step size with which the preference for an algorithm is increased. The received reward R is also used to update the reference reward level $\bar{R}$ to $\bar{R}+\alpha(R-\bar{R})$, again with a parameter $\alpha \in (0, 1]$ governing the adaptation speed.

Random Selection To check the effectiveness of multi-armed bandit policies, they should be compared to a policy that does not converge. Such a simple scenario is implemented by a random selection policy — RandomSelection — that just picks a random arm for each round. In the long run, its overall average reward $\hat{R} = \frac{\sum_{i=1}^{k} \hat{R}_i}{k}$ will therefore converge to the average algorithm performance on the problem; the same holds for the variance. This situation corresponds to a manual algorithm selection by a user without prior knowledge on the algorithms in $\mathbb{A}$.

The policies construct their selection mappings from the rewards they receive (see sec. 2.3.2, p. 32). The mappings are encoded by preference values (RewardComparison), selection probabilities (SoftMax), or arm indices (UCB). A policy that outperforms RandomSelection after a certain number of rounds has constructed a selection mapping $S \in \mathbb{S}$ that is average-effective in the sense of definition 2.1.5 (p. 17). Here, the problem set P only contains a single element: the simulation problem to be replicated. Since $|P| = 1$, no policy can be adaptive-effective (see def. 2.1.6, p. 18). The best constant selection mapping S_C^* (see def. 2.1.8, p. 19) selects the algorithm with the truly best average reward right from the beginning; it does not waste any resources on performance exploration. Hence, it outperforms any adaptive policy. By comparing the performance of a policy with such an optimal strategy *and* RandomSelection, the potential performance impact of the adaptive simulation runner can be assessed.

Potential Enhancements for Large-Scale Experimentation

Currently, a policy is reset for each new simulation problem that is encountered, i.e., it starts out without any prior knowledge. If an experiment comprises many rather *similar* simulation problems, e.g., generated by an optimization algorithm, it might be beneficial to transfer the previously gathered knowledge to new problems. While an off-line mechanism that supports such knowledge transfer is discussed in section 7.2.2, there are specific policies for such *nonstationary* multi-armed bandit problems (see discussion in [298, p. 38–39]).

Furthermore, the adaptive simulation runner is currently implemented on top of the parallel simulation runner presented in [201], which replicates one simulation problem after another. While this execution order is fine in most cases, it may hamper the performance of the adaptive simulation runner when applied to experiments that use many computational resources in parallel and comprise of several simulation problems, e.g., simulation-based optimization or validation experiments carried out on a cluster. This is because too much parallelism during the replication of a simulation problem will hamper the effectiveness of the adaptation policies. A good decision requires knowledge on prior rewards, whereas increasing the number of available processors forces the policy to make more decisions based on less knowledge. The problem is illustrated in figure 7.9. Instead of processing simulation problems sequentially, each being replicated with all available resources (fig. 7.9, left), it is now much more beneficial to parallelize the execution on the problem level (fig. 7.9, right), i.e., to execute the first replication of all simulation problems in parallel, then the second one, and so on. While such a scheme ensures ideal conditions for the adaptive replication policies, it may hamper the efficient execution of experiment steering methods on top. For example, a simulation-based optimization method may require some problems to be processed before it is able to generate new ones. Future work on the adaptive simulation runner should account for this issue and develop more advanced schemes to control execution order.

7.2.2. Simulation Algorithm Portfolios

Apart from issues in the presence of many computational resources (see fig. 7.9), there is another (more serious) obstacle for using the adaptive simulation runner. It is directly related to the characteristics

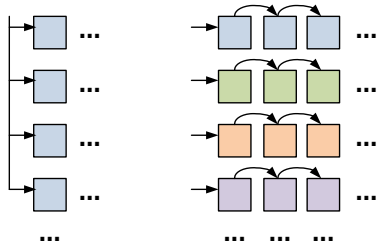

Figure 7.9.: Execution order matters: replicating a single simulation problem on many processors in parallel (blue, left) forces a policy to make many uninformed decisions. It would be advisable to sequentially replicate several problems in parallel (right).

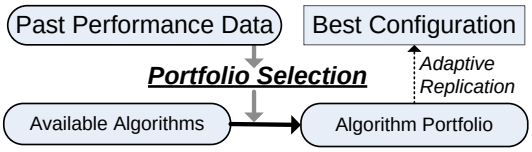

Figure 7.10.: Portfolio selection for the adaptive simulation runner.

of JAMES II: since its principal design goals are flexibility and extensibility, the number of available runtime configurations to solve a single setup — i.e., the size of $\mathbb{A}$ — may become quite large (see sec. 4.3, p. 89). For many simulation problems, the number of available algorithms may thus exceed (or be comparable to) the number of required replications. An obvious solution is to *restrict* the number of options from which the policy may choose, i.e., the number of algorithms to be considered. As discussed in section 2.5.2 (p. 40), the task of identifying algorithms that are *likely* to perform well can be regarded as selecting an algorithm portfolio (see fig. 2.9, p. 41). The portfolio should consist of algorithms that have proven to be useful before, while algorithms that performed badly so far should be dismissed. In other words, portfolio selection mechanisms consider past performance data to restrict the number of arms a policy may choose from. This section discusses the portfolio selection mechanism that has been implemented to augment the adaptive simulation runner. Its aim is to broaden the applicability of the adaptive simulation runner to problems where the multitude of eligible runtime configurations otherwise hampers convergence within the given number of replications.

Applications and Requirements of Simulation Algorithm Portfolios

Besides the reduction of options to be considered by multi-armed bandit policies, there are several other potential application domains for simulation algorithm portfolios. As discussed in section 2.5.2 (p. 40), they *simplify* the ASP; hence, they could be used as a preprocessing step for selector generation (see sec. 6.2.3, p. 128). Narrowing down the choice would be particularly helpful for problem classification (see fig. 6.2, p. 116), which then has much less categories to choose from and thus a much simpler problem to solve.

Moreover, simulation algorithm portfolios can be used to rank algorithms. This could be interesting for performance analysts and developers alike. If an algorithm is selected for many kinds of portfolios, e.g., portfolios for different modeling formalisms, this underlines its importance for overall system performance and may motivate further code improvements. By comparing selected portfolios of different sizes $s = 1, \ldots, k$, one may also examine how many *relevant* runtime configurations need to be considered for a problem domain, i.e., this hints at the number of distinguishable problem classes.

Another application of simulation algorithm portfolios could be the parallelization of simulation runs in case peak performance is crucial. If runs are parallelized in a brute-force manner, i.e., the results of the first one are used while the others are dismissed, it is important to narrow down the algorithms that are most likely to come in first. The size of such portfolios would hence be limited by the number of threads that can be executed in parallel, e.g., the number of cores on a CPU.

Two potential application domains of simulation algorithm portfolios — algorithm ranking and brute-force parallelization — require to specify the desired portfolio size manually. The same holds when portfolio selection is used as a pre-selection mechanism for replication policies: the most suitable size of a portfolio is likely to depend on the number of replications that are to be executed. In case

only a few replications are required, using a large portfolio may still hamper the convergence of the policy. On the other hand, a large number of desired replications may warrant the selection of a larger portfolio — to increase the chance that it contains the algorithm that is truly the most suitable one. As Gomes and Selman show in [114], changing the desired portfolio size may strongly affect the composition of efficient portfolios, i.e., the shape of the efficient frontier (see sec. 2.5, p. 38). A simple illustration of this effect is depicted in figure 7.11.

Another factor that influences portfolio structure is the objective function to characterize the desired properties of a portfolio. Many methods for financial portfolio selection are based on Markowitz' formulation of the problem (see eq. 2.21, p. 40). In contrast, the selection of simulation algorithm portfolios may employ custom objective functions that are tailored to the specific application domain. For example, portfolios for brute-force parallelization should contain one very fast algorithm for each problem class to be encountered — it does not matter how well the other algorithms perform, as long as *one* algorithm performs well.

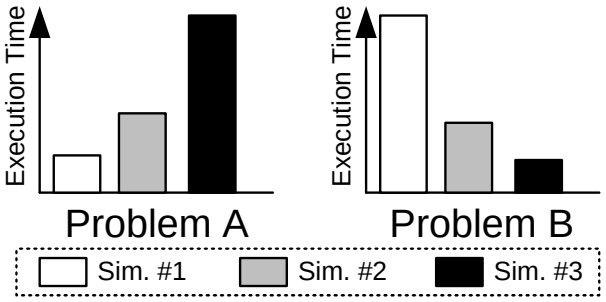

Figure 7.11.: Portfolio size matters: a single-element portfolio would contain simulator 2, whereas a two-element portfolio may contain simulators 1 and 3 instead.

Similarly, the portfolio vector $\vec{\alpha} \in [0,1]^k$ ($|\mathbb{A}| = k$, see sec. 2.5.1, p. 39) that denotes the investments per asset is not always meaningful. This way of defining a portfolio is, for example, rather pointless in case of adaptive replication. Most multi-armed bandit policies do not consider any weights, e.g., the UCB policies try out every single arm at first. As a policy shall identify the most suitable option on its own, portfolio selection just has to decide which arms to choose from — which algorithm belongs to the portfolio and which does not. In other words, the search space for the optimization is reduced to $\{0, \frac{1}{s}\}^k$, where s is the again the portfolio size, i.e., $s(\vec{\alpha}) = |\{\alpha_i | \alpha_i \neq 0\}|$. Instead of dealing with an uncountable set $[0,1]^k$, the search space now only has 2^k elements.

Other requirements may complicate the search for a good simulation algorithm portfolio. For example, users may want to consider *multiple* criteria, e.g., algorithms that are fast and do not require too much memory. As discussed in section 2.5.4 (p. 46), this leads to further difficulties. Finally, portfolio selection techniques should not require too much computational resources. As discussed above, user criteria, objective function, and size constraints may all vary — hence portfolio selection should be done *on demand*, e.g., right before the experiment starts and the adaptation policies are applied. This places portfolio selection onto the critical path, i.e., the longest sequence of computations that has to be executed sequentially, no matter how many resources are available (see sec. 3.3.1, p. 71). Portfolio selection is executed just *before* the replications can be computed in parallel, thus it should not take too much time.

Implementation Details

A new plug-in type for algorithm portfolio selection is defined for JAMES II, so that different selection mechanisms can be exchanged and compared with each other. Plug-ins implement the `IPortfolioSelector` interface, which supports the general definition of portfolios (i.e., $\alpha_i \in [0,1]$) in order to be applicable in different scenarios. The problem description handed over to the portfolio selector is represented by objects of type `PortfolioProblemDescription`. A description contains the past performance data to be used, specifies minimum and maximum portfolio size, and also the risk aversion factor λ (see sec. 2.5.1, p. 39). The problem description may contain data for more than one performance metric. Since some performance metrics shall be maximized while others shall be minimized, the description also contains flags to distinguish both cases for each metric. For example, one may want to maximize the accuracy while minimizing the execution time. Any portfolio selector

is able to search for both maximizing and minimizing portfolios; a simple inversion of the performance data (multiplying the past performances by -1) is carried out automatically.

Additional constraints like limiting the portfolio size make this problem hard to solve (see sec. 2.5.1, p. 39). As a simple and general benchmark algorithm, a plug-in that randomly searches the constrained portfolio space and uses equation 2.21 (p. 40) for optimization has been implemented. The development and evaluation of techniques that are actually able to consider multiple metrics are subject to future work. Since more advanced meta-heuristics like genetic algorithms have been successfully applied to constrained portfolio selection in finance [234, 331], a similar portfolio selection mechanism is implemented for the adaptive simulation runner.

A Genetic Algorithm to Select Simulation Algorithm Portfolios

Genetic algorithms (GAs) — already briefly discussed in section 2.4.2 (p. 35) — are usually straightforward to implement. They basically require to define:

- An *encoding* for all potential solutions, e.g., as a tuple of some type like `boolean`. GAs regard solutions as individuals; the encoding represents their *genome*.
- A *fitness function* $\phi(\vec{\alpha})$ to assess a potential solution. This function can be regarded as an objective function (see sec. 2.1.3, p. 20).

Additionally, some termination conditions can be given, e.g., the maximum amount of wall-clock time the execution may take. The algorithm as such is quite simple: at first, an initial generation g_0 is created, e.g., by randomly drawing potential solutions. Then, the fitness of all individuals in g_0 is assessed by the fitness function ϕ. An individual's relative fitness (compared to the fitness of other individuals) determines the probability with which it will be selected for reproduction. The genomes of reproducing individuals are combined to create offspring, thereby imitating genetic processes. The offspring forms a new generation, g_1, of the same size as g_0. The fitness of the individuals in the new generation can now be assessed by ϕ, which again leads to subsequent selection and reproduction, resulting in generation g_2, and so on.

Encoding and Genetic Operations The reproduction mechanism of the implemented GA combines the genomes of two individuals to produce two new individuals as offspring. This is done by *recombining* the parent genomes and applying a *mutation* operation afterwards. Since portfolio selection is so far focused on enhancing the adaptive simulation runner, the encoding does not include any weights. A portfolio of algorithms can therefore be represented as an array of booleans. However, this may hamper the scalability with respect to $k = |\mathbb{A}|$, because for large sets of algorithms and small portfolio sizes these arrays are sparse and large. Portfolios are thus encoded as lists of fixed length instead, where the length corresponds to the maximal portfolio size. Each field contains the index of an algorithm that belongs to the portfolio, i.e., all algorithms are enumerated with $1, \ldots, k$. The numbering is arbitrary but fixed during the portfolio selection process. If the portfolio size is less than maximal, the last 'slots' in the list remain empty.

The recombination procedure selects a random point at which two lists containing algorithm indices are split and recombined, i.e., the tail of one is added to the head of the other. This approach is sketched in figure 7.12 (p. 144). Algorithms contained in both portfolios have to be left out in this operation, to avoid duplicates (e.g., algorithm 4 in fig. 7.12). Otherwise, the results of a recombination may violate the constraints regarding minimal portfolio size: a portfolio is a *set* of algorithms, so if its genome contains duplicate algorithm indices, the actual portfolio size is smaller than the list size.

Mutation is important to explore formerly unknown solutions, e.g., to include algorithms that have not been included in a portfolio before. It randomly selects an algorithm from the portfolio and replaces it with another one that has not been included yet (see fig. 7.12). The overall functioning of GA-based portfolio selection for adaptive replication is sketched in figure 7.13 (p. 144).

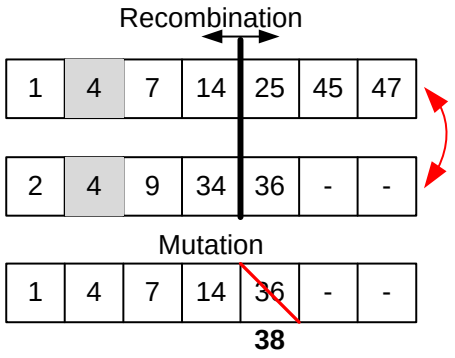

Figure 7.12.: Recombination and mutation for GA-based portfolio selection. If an algorithm is part of both portfolios to be recombined (e.g., algorithm number 4 above, box marked gray), it is left out of the recombination procedure and later added to both offspring portfolios.

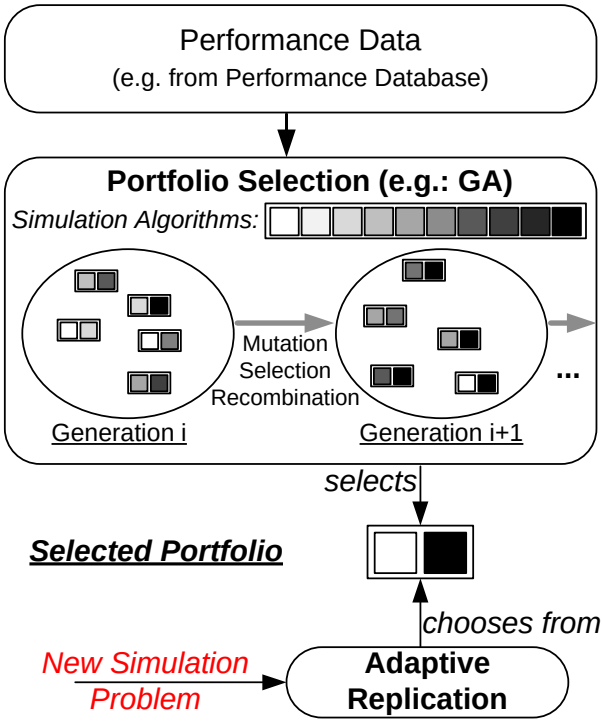

Figure 7.13.: GA-based portfolio selection and adaptive replication.

Fitness Function The fitness function used by the GA can be exchanged arbitrarily, to account for the different application domains of simulation algorithm portfolios. Currently, it is tailored to adaptive replication: the fitness $\phi(\vec{\alpha})$ of a portfolio $\vec{\alpha}$ sums over all considered simulation problems from the problem space $\mathbb{P}$ (def. 2.1.1, p. 14), $P_1, \ldots, P_n \in \mathbb{P}$, each time focusing on the *best-performing* algorithm of the portfolio.

This notion of portfolio quality assumes that the adaptive simulation runner is able to find the best algorithm of the portfolio during replication. However, a convergence in time — i.e., within the number of desired replications — can usually not be guaranteed[6] and might be prolonged or prevented by too many options or a lock-in (see sec. 2.3.2, p. 31). The user's risk preference λ (see sec. 2.5.1, p. 39) is therefore used to define the trade-off between the expected performance in case of a successful convergence (left-hand side, risky) and the best average portfolio performance ($\bar{p}(\vec{\alpha})$, secure):

$$\phi(\vec{\alpha}) = \lambda \frac{\sum_{i=1}^{n} \min\{p_{i,j} | \alpha_j \neq 0\}}{n} + (1 - \lambda)\bar{p}(\vec{\alpha}) \tag{7.6}$$

where

$$\bar{p}(\vec{\alpha}) = \frac{\sum_{i=1}^{n} \sum_{\alpha_j \neq 0} p_{i,j}}{n \cdot s} \tag{7.7}$$

and $p_{i,j} \in \mathbb{R}^+$ is the performance observed for algorithm a_j on problem P_i. The $p_{i,j}$ are currently not normalized to the hardness of the given problem, e.g., via $p'_{i,j} = \frac{p_{i,j}}{min_k(p_{i,k})}$, so that the selection is biased towards problems that require more wall-clock time for their solution. The expression for $\bar{p}(\vec{\alpha})$ averages the performances of all algorithms in the portfolio. It represents the expected performance of

[6] While zero-regret policies do guarantee to *eventually* converge to the optimum, it is uncertain how many rounds this takes.

randomly selecting algorithms from the portfolio, instead of converging to the best one. Optimizing it can be regarded as a risk-averse, since its performance is relatively independent from the performance of the replication policy.

Current Limitations While the current implementation of the GA-based portfolio selector yields promising results (see ch. 9, p. 177), there are several limitations that should be addressed in future work. First of all, only single performance metrics are considered so far, although equations 7.6 and 7.7 are straightforward to generalize towards multiple metrics. Additionally, the GA is currently implemented as a purely sequential algorithm. Current portfolio selection problems in JAMES II are small enough to be solved in little time (less than 1 s on a desktop computer). However, larger portfolio selection problems — i.e., problems with more algorithms to choose from or larger amounts of performance data to consider — may motivate the implementation of parallel variants. The issues with problem size also hint at another, more serious problem: as already outlined in figure 7.10 (p. 141), portfolio selection requires performance data. The amount of data required for a good selection can be considerable, as it grows with the number of available algorithms. The data is also challenging and time-consuming to obtain (see ch. 3, p. 61). While it is an interesting direction of future work to investigate how *much* performance data is really necessary to obtain portfolios of a certain quality, one may also improve the situation by speeding up and automating the performance evaluation of simulation algorithms as such. Developers and performance analysts would also profit from such a mechanism, which is introduced in the next section.

7.3. Automated Runtime Performance Exploration

Both performance data mining (ch. 6, p. 115) and portfolio selection (sec. 7.2.2, p. 140) require large amounts of performance data. This data should be collected automatically — otherwise, the considerable effort of conducting experiments will hinder the application of algorithm selection methods in practice.

An experiment to collect performance data can be regarded as searching through the performance space of simulation algorithm(s): the unknown performance function $p : \mathbb{A} \times \mathbb{P} \to \mathbb{R}^n$ (see def. 2.1.1, p. 14) needs to be evaluated for *some* algorithms and *some* problems, in order to infer more general findings (e.g., with the help of machine learning). This requires to carefully select which algorithms $A \subseteq \mathbb{A}$ to apply to which problems $P \subseteq \mathbb{P}$. Vuduc et al. recognize this *"[...] process of searching itself as an interesting and challenging problem"* [318, p. 125].

Both the set of algorithms, $\mathbb{A}$, and the set of simulation problems, $\mathbb{P}$, may be quite large. However, in contrast to empirical tuning (see sec. 2.7, p. 54), where the algorithm space might be too large to handle [319], the number of simulation algorithms is typically much smaller than $\mathbb{P}$. Consequently, most methods described here focus on the selection of suitable elements from $\mathbb{P}$ and regard the set of algorithms as given. Section 7.3.1 discusses the role of benchmark models in this context. It turns out that a careful construction and configuration of such models is a prerequisite for any meaningful performance analysis. Section 7.3.2 builds up on these findings and introduces a mechanism to automatically calibrate the simulation end time.

This enables the implementation of several components that interact with each other to *automatically* explore the performance space of simulation algorithms. This *simulation (performance) space* $\mathbb{P} \times \mathbb{A} \times \mathbb{R}^n$ is characterized by the performance function $p : \mathbb{A} \times \mathbb{P} \to \mathbb{R}^n$, defined for the ASP (cf. fig. 2.1, p. 14). The exploration system is described in section 7.3.3; a sample application to support simulator development is outlined in section 7.3.4.

7.3.1. Benchmark Modeling

The importance of selecting suitable problem instances for algorithm performance analysis and comparison is well-established [101, 325]. Yet, there are still many pitfalls to benchmarking [163], also in

the field of modeling and simulation. After discussing some related work, e.g., regarding benchmarks considered in other algorithm selection studies, this section argues for the use of dedicated benchmark models to solve the ASP. It concludes with outlining the most important properties any benchmark model should have and discusses two sample benchmark models, one for SSAs and one for parallel and distributed discrete-event simulation.

Related Work

Well-established models to benchmark simulation algorithm performance are rather rare. While there have been approaches to introduce general benchmarks — e.g., for specific modeling formalisms [110], specific simulation algorithms [12, 89], or specific application domains [31] — only few of them have been widely adopted (a notable exception is the PHOLD model, see p. 149).

As using common input instances is a prerequisite for reproducibility (see sec. 3.1.2, p. 63), the lack of established benchmark models complicates the execution of sound performance studies. A lack of reproducibility also prevents others to uncover software bugs or faulty setups — which may easily arise [115, 226, 294] — so that the scientific significance of obtained results is diminished. Other computer science communities resolved these problems by establishing a common representation of problem instances and curating databases of sample problems, e.g., the UCI machine learning repository [9] or SATLIB [143]. The latter includes particularly hard problem instances that were chosen by considering a phase transition (see sec. 2.7, p. 56).

Besides realistic and synthetic (i.e., artificial) benchmarks, the performance evaluation community mainly distinguishes between macro- and micro-benchmarks. Micro-benchmarks are small and relatively simple programs to analyze a particular aspect of the system under study [290]. They usually are synthetic, i.e., specified for the purposes of the performance analysis. Macro-benchmarks, in contrast, subsume real-world applications or application kernels (i.e., the most characteristic parts of applications), but also recorded application execution traces and synthetic benchmarks of a certain size [4, 290]. Synthetic macro-benchmarks are constructed to mimic relevant aspects of real-world applications, i.e., they shall *"[...] strike a balance between reproducibility and relevance"* [290, p. 6]. They can be derived from micro-benchmarks [4] and are useful for exploring realistic scenarios by changing parameters [289].

This broad categorization of benchmark types can be roughly translated to simulator performance evaluation: real-world models can be regarded as macro-benchmarks (i.e., applications), whereas simple benchmark models — e.g., developed to determine the cost of a certain operation — can be regarded as micro-benchmarks. The latter are useful to study the performance impact of certain phenomena, e.g., roll-backs in an optimistically synchronized parallel discrete-event simulation (see sec. 1.3.2, p. 6). Other kinds of benchmark input — such as traces — are seldom used in evaluating simulators, even though traces of parallel and distributed discrete-event simulation can also be analyzed (see trace-based analysis, discussed in sec. 3.3.2, p. 73). While traces of distributed simulations may become quite large, there are additional techniques to extract their most important properties via sampling and data analysis [26]. Even the construction of benchmarks can be automated to some extent, e.g., by dedicated tools like CodeMRI [4] (for measuring file system performance).

There are entire journals (e.g., [252]) devoted to performance evaluation — yet there seems to be no particular benchmarking methodology for simulation algorithms, i.e., no specific methodology to construct suitable benchmark models. General advice, on the other hand, is abundant: Small et al. propose to limit the impact of random elements, to focus benchmark construction on the aspects of interest, and to make the benchmark representative of many problem classes by introducing additional parameters [290]. The last suggestion is also maintained by Hooker, who states: *"Rather than agonize over whether a problem set is representative of practice, one picks problems that vary along one or more parameters"* [142, p. 202].

However, there is some — justified — skepticism when it comes to performance evaluation with synthetic benchmark problems, e.g., regarding their representativeness [78, p. 298]. Similarly, Weihe argues in [325] that the benefits from using randomly generated problem instances are often unclear.

These issues give rise to the question of whether it is beneficial to rely on synthetic problem instances at all — why not restrict analysis to real-world problems?

Advantages of Synthetic Benchmark Models

Relying on synthetic benchmarks with some random elements is often the only way to carry out a thorough performance evaluation. This is particularly true for simulation algorithms that process models of a *new* formalism, i.e., when there are only few sample models available.

Furthermore, the usage of real models brings some problems in its own. While such models are clearly representative of the problem domain *now*, this does not mean that the simulation algorithm under scrutiny will also have to cope with the same kind of model *in the future*. Future models might be, for example, considerably larger. This is one reason why performance analyses of simulation algorithms often consider their scalability; a property which is of particular interest in parallel and distributed simulation, since it is often motivated by the simulation of ever more complex models. Conducting a scalability study with real models would require several models with similar properties but different size. These may not be easy to find — and similar problems in finding suitable models may occur for all other kinds of performance experiments as well.

Another problem of using real models for performance evaluation is their complexity. Such models can hardly be described in the publication that presents the performance results, nor can they be easily recreated by others for reproducing the experiments. In contrast, synthetic benchmark models that are defined carefully exhibit the following desirable properties:

- **Parameterizability**: Many real-world models have parameters to adjust them to the scenarios of interest. These parameters usually affect a model's behavior and therefore also the performance of the simulation algorithm — but only indirectly. In contrast, the parameters of synthetic benchmark models should be *specifically designed* to explore all performance-relevant behaviors that models of the given type may exhibit. Although this often results in large parameter spaces, it allows to directly investigate the impact of particular performance-relevant model properties on algorithm performance. This simplifies data analysis and increases the expressiveness of performance experiments. Well-chosen benchmark model parameters may hence allow to identify the *features* of all models a simulation algorithm works well on, e.g., for the later application of algorithm selection techniques.

- **Scalability**: Real-world application models are often not scalable enough for thorough performance studies — a problem that is closely related to the issues with parameterization. This is of particular importance when studying new algorithms for parallel and distributed simulation, as these may only show advantageous performance for models that are large enough. Moreover, some interesting phenomena, e.g., the effects of caches [191], do not occur when models are too small. Automated performance analysis could address scalability studies in various ways, e.g., by providing simple means for setting up the experiments, or for statistically analyzing the growth of the measurements under scrutiny.

- **Simplicity and Comparability**: Real-world application models tend to be rather complex and intricate. In contrast, a synthetic benchmark model should be *as simple as possible*. This reduces the error potential when implementing or specifying the model for the setup at hand. A simple model may also enable performance analysts to gain some theoretical insights into its behavior. It facilitates debugging, simulator validation, and interpretation of performance results. However, the major benefit of simplicity is comparability (see fig. 3.1, p. 62), which also motivates the re-use of established benchmark models. It allows to compare performance across different simulation systems, hardware platforms, and algorithms. To ensure that performance results are indeed comparable, it is therefore best to keep benchmark models simple.

- **Quasi-Steady State**: The above properties of benchmark models are rather obvious and often followed implicitly — in fact, they are not specific to modeling and simulation, but apply to

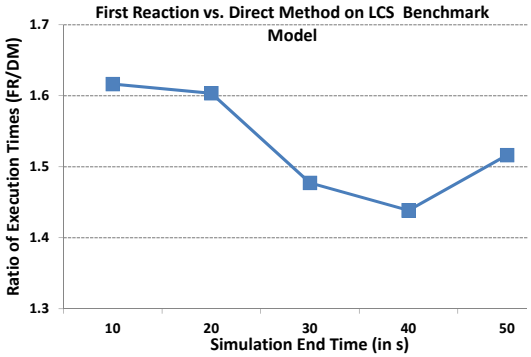

Figure 7.14.: Bias due to lack of quasi-steady state. The Linear Chain System benchmark model [31, p. 4062] for SSA (see ch. 9, p. 177) does not exhibit a quasi-steady state. Hence the overhead of one simulation algorithm with respect to the other varies from 61% to 43% for a single setup, depending on the simulation end time.

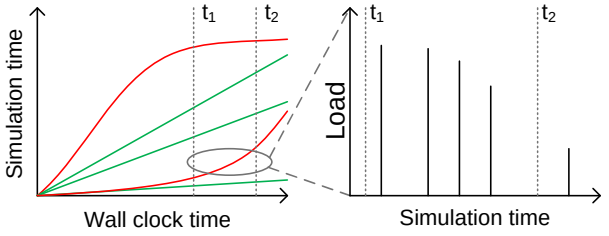

Figure 7.15.: Models need to be in a quasi-steady state (w.r.t. computational load) if the wall-clock time shall grow linearly with the simulation time interval. If not (see red trajectories in left plot), this means that the computational load of the model changes over time (right plot).

experimental algorithmics in general (e.g., [163]). However, there is another important property that is simulation-specific and less obvious: in case execution time performance is measured, a benchmark model should be in a *quasi-steady state*. This means it shall exhibit a *steady* behavior regarding the computational load it imposes on the simulator in a given simulation time interval.

Real-world models often have warm-up phases and therefore do not comply to this property, so that simulation time does *not* grow linearly with wall-clock time — there even might be *different* phases in model execution that are advantageous for *different* simulators. The arising uncertainty about the current state of the model, and hence the workload a simulation algorithm is confronted with, may be a strong source of bias in experiments focused on execution time. A real-world example of this effect is shown in figure 7.14.

The cause for this is sketched in figure 7.15: the time at which the simulation is stopped affects which algorithm is deemed to be the fastest. If, for example, two discrete-event simulators are compared, one might work best when encountering relatively few but complex events, e.g., until t_1 is reached. Afterwards, the load characteristics of the model change and at time t_2 the second algorithm might be faster. In such situations, execution time comparisons are essentially futile. Since workload characteristics and their impact on simulator performance are of particular interest in many performance studies, such changes in load should be made explicit.

Therefore, it is advantageous to construct models that remain in very similar states throughout the execution, a property that can often be achieved by introducing additional parameters. While larger parameter spaces require more efforts to explore, a model's quasi-steady-state behavior also facilitates the usage of automatic mechanisms for simulation time *calibration* (see sec. 7.3.2, p. 150), which in turn may speed up performance analysis. Note that the state only needs to be relatively steady with respect to the computational load characteristics, as depicted in figure 7.15.

Nevertheless, it should be noted that all performance studies, however interesting their results may be, are worth nothing unless at least *some* of the benchmark models are chosen so that they *represent* real-world problems. In other words, the parameter space of the benchmark models has at least to include model size, structure, and behavior that can also be found in comparable real-world models. This is hard to achieve and usually involves application-specific surveys, as well as conjectures regarding future model properties. Representativeness is crucial for the interpretation

of the results. Consequently, lack of representativeness is an often-raised criticism of performance analyses [96, 163, 290]. Ideally, one would address this problem by additionally evaluating real-world models for the validation of the general findings.

Example Models

Two benchmark models — one for each exemplary problem domain, SSA and PDES (see sec. 1.3, p. 4) — shall exemplify the realization of the aforementioned properties.

CCS The *cyclic chain system (CCS)* model was developed for a JAMES II performance study on the performance of SSA algorithms [158]. The main structure is rather straightforward. All reactions are defined as:

$$R_i : \sum_{j=0}^{k} S_{(i+j) \pmod N+1} \xrightarrow{c} \sum_{j=k+1}^{2k+1} S_{(i+j) \pmod N+1},\ i \in [1, N], k \leq N \tag{7.8}$$

where N is the overall number of species $S_1, \ldots, S_N$, and k defines how many species are involved in a reaction. The rate constant c is equal for all reactions R_i. The system is *cyclic* (note the modulo operator in eq. 7.8) and there are as many reactions as there are species.[7] The overall number of particles in the system always stays constant, because each reaction produces as many elements as it consumes. A CCS model starting with all species having the same amount of particles is always forced back into this equilibrium state — it hence exhibits the quasi-steady-state property.

The model is also parameterizable with respect to the interdependency of reactions (k) and the model size (N). The degree of interdependence among reactions is of interest because some algorithms follow the approach by Gibson and Bruck [104] and exploit the independence of reactions: they first construct a dependency graph and then restrict updates to dependent reactions, in order to speed up the simulation. By controlling the parameter k, one may now study *at which point* — i.e., which k — such a strategy becomes beneficial. Clearly, the construction of a dependency graph in case *all* reactions are interrelated ($k \geq \lfloor N/3 \rfloor$) does not yield any performance gain, it just induces additional overhead. This example shows how the construction of benchmark models may account for the optimizations carried out by the simulation algorithms to be evaluated. Since N and k (depending on N) can be chosen arbitrarily, the CCS model is clearly scalable. Its homogeneous structure, specified by equation 7.8, also makes it fairly simple to implement. The CCS thus fulfills all desirable benchmark model properties; it is applied in the case study on SSAs presented in chapter 9.

PHOLD The *PHOLD* model [89] is an established synthetic benchmark model for parallel and distributed discrete-event simulation. It is widely used to evaluate new PDES implementations (e.g., [17]). Its compliance with the four design principles (parameterizability, scalability, simplicity and comparability, and a quasi-steady state) may explain its widespread use. It is derived from the HOLD benchmark for event queues [164] and very simple to implement: a set of model entities, i.e., logical processes (LPs, see sec. 1.3.2, p. 6), exchange time-stamped events at random. The total number of events in the system is fixed. If a model entity receives an event, it creates a new event with a future time stamp and sends it to a randomly chosen neighbor. Since each event generates one new event and message destinations are random, the communication pattern as well as the number of events to be managed stays roughly the same over time. PHOLD is therefore in a quasi-steady state. It is also scalable in at least two dimensions: increasing the number of events will increase the amount of model-inherent parallelism, so that model parallelism can be set into direct relation to the performance of parallel and distributed simulators. Increasing the number of model entities increases the memory footprint of the model. PHOLD is parameterizable in many other aspects. For example, the topology of the model can be adjusted, i.e., which LP has which neighbor LPs. The same goes

[7] In fact, there is yet another parameter that defines *how many* identical reactions R_i there are per species (to investigate scalability w.r.t. reactions, see fig. 9.1, p. 182), but this does not interfere with the properties of the model.

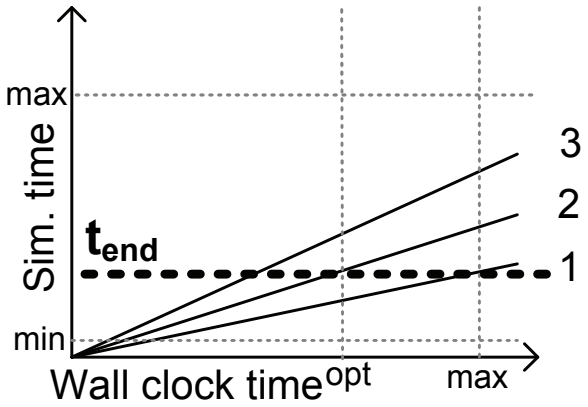

Figure 7.16.: Calibration means to identify a simulation end time so that algorithms of differing speed (1: slowest, 3: fastest) finish execution after $\approx t_{opt}^{wct}$ all clock time. The quasi-steady state property of the model is presumed, so the relationship between wall-clock time and simulation end time is *linear*.

for the probability distribution to create future event time stamps, or the synthetic computational load each event imposes on the simulator. PHOLD is applied in the case study on PDES presented in chapter 10.

Very much like realistic models, synthetic benchmark models have to be used with some care. Before conducting any serious study, it has to be analyzed if the parameters of the model actually allow to investigate the aspects of interest. For example, PHOLD as such does not lend itself to studies on dynamic load balancing algorithms (see sec. 1.3.2, p. 7), caused by its *quasi-steady state* property: the computational load of the model does not change over time, and there are no preferential communication patterns that can be exploited by LP migration. It is always the same number of events that is propagated through the same neighborhood. Consequently, PHOLD needs to be adjusted when algorithms that rely on exactly this kind of model dynamics — such as load balancers — shall be studied. For example, a subset of nodes with additional behavior could be added, as done in [212]. Note that the benchmark model may still retain the quasi-steady state behavior for larger *intervals* of simulation time, e.g., by controlling model dynamism with fixed parameters for frequency and amplitude of load changes.

7.3.2. Simulation End Time Calibration

After having constructed a suitable benchmark model (sec. 7.3.1), the next step to enable automated experiments on runtime performance is to pick the right *configurations* of the benchmark model, in terms of parameterization. While a systematic automated exploration of a benchmark model's parameter space (e.g., by using meta-modeling, see sec. 3.2.2, p. 68) is still work in progress, experimentation is even challenging when drawing random elements from a parameter space, or evaluating parameterizations given by a performance analyst. This is because the *hardness* of the benchmark model configuration, i.e., the computational load that is required to solve it, varies strongly among model configurations. This makes it necessary to control the interval of simulation time for which a benchmark model shall be simulated.

Imagine a performance analyst who wants to compare two simulation algorithms on a range of different problems, all represented by differently parameterized instances of a benchmark model. The analysts sets the simulation time interval for which all setups shall be executed to a duration of 10 seconds (simulation time, see fig. 3.2, p. 69). Now consider a benchmark model configuration which is very small and simple to compute. The execution time of *both* algorithms will be rather short, maybe in the region of milliseconds only (wall-clock time). If run times are that short, bias introduced by

hardly controllable factors (operating system or user operations, prior state of the hardware) is likely to have a considerable impact, rendering most performance comparisons meaningless. A better option would have been to increase the size of the simulation time interval, e.g., to 100 or 1000 seconds (simulation time).

On the other hand, the next benchmark model setup to be explored might be very hard to simulate, i.e., both algorithms require much time to solve it — even for just 10 seconds of simulation time. While long execution times are safer to use for comparison, they also require large amounts of resources. Instead of observing that algorithm A is twice as fast as algorithm B on the given problem by letting them run 2 and 4 *hours*, respectively, the same information could perhaps also be inferred by letting them run 2 and 4 *minutes*? In such cases, a better option would be to decrease the size of the simulation time interval, e.g., to 1 or 0.1 seconds (simulation time).

These examples illustrate that the simulation time interval for which a set of benchmark model configurations shall be simulated needs to be *adapted* to each individual model configuration, otherwise the performance analyst is likely to end up with either meaningless results or a very time-consuming experiment (or both). This is particularly true if no good prediction regarding the general hardness of a benchmark model configuration is available.

An adaptive scheme may vary the size of the simulation time interval by keeping the start of the interval fixed and selecting a suitable simulation end time, t_{end}, for each benchmark model configuration. Figure 7.16 (p. 150) shows the basic problem that simulation end time calibration aims to solve: given a set of simulation algorithms, how to set the simulation end time t_{end} in a manner that the *average* execution time of the algorithms is approximately t_{opt}^{wct}? The time t_{opt}^{wct} would be the optimal wall-clock time a simulation algorithm requires to solve the simulation problem, i.e., to simulate the given benchmark model configuration for the given simulation time interval. It should be set to the least time for which the impact of noise, e.g., by the operating system, is deemed negligible (see sec. 3.2.1, p. 66). Since this depends on the execution environment, it should be decided by the performance analyst. The analyst also has to decide upon the interval $[t_{min}, t_{max}]$ in which the simulation end time t_{end} should lie. The lower border, t_{min}, needs to be large enough to assume a quasi-steady state. For example, a PHOLD model extended by oscillating elements (e.g., to account for load heterogeneity, as discussed in sec. 7.3.1, p. 149) should be configured with a t_{min} that ensures the execution of several oscillations.

Another simple approach to limit algorithm execution time is to simply stop it when it exceeds the maximum wall-clock time one is willing to invest. This technique can be combined easily with simulation end time calibration, and is also used by others who aim at solving the ASP (e.g., [205, p. 38]). Gagliolo and Schmidhuber link this method to the notion of *censoring* in demography [96, p. 302]: depending on the analysis, the fact that an algorithm is *not* able to finish in time may also be considered as a performance result.

A Simple Calibration Algorithm

The simple calibration algorithm introduced in the following only considers a sample set of algorithms $A \subseteq \mathbb{A}$ for searching a suitable simulation end time. The sample can be manually defined by the user, e.g., based on some general findings on algorithm performance, or by using portfolio selection (see sec. 7.2.2, p. 140). Otherwise, the user may specify the plug-in types which are deemed to have most performance impact. For example, the choice of event queue may be more important than the choice of a random number generator, performance-wise. A random sample is then constructed by preferring algorithms comprising different plug-ins of the predefined types.

Each algorithm is then applied once to the current benchmark model setup. If the algorithms are sufficiently representative for $\mathbb{A}$, averaging over their execution times should result in a rather good estimate of the overall average execution time. The sample size is yet another parameter a performance analyst may set. Some pseudo-code that represents the core of the calibration algorithm is outlined in listing 7.1 (p. 152).

The basic idea is that, since the benchmark model should have the quasi-steady state property for

```
public void calibrate(double[] durationsWCT) {
2   //Initialization [...]

4   //Calculate average WCT over all algorithms in the sample
    runtime_wct = avg(durationsWCT);
6
    //Check if current simulation end time is better then the best end time found so far
8   if(|bestAvgWCTime - t_opt^wct| > |runtime_wct - t_opt^wct|) {
      bestEndTime = currentSimEndTime;
10    bestAvgWCTime = runtime_wct;
    }
12
    //Check if calibration is finished since achieved WCT is close enough to optimum
14  if(runtime_wct ∈ [t_opt^wct · (1 - θ), t_opt^wct · (1 + θ)])
      done = true;
16
    //Search by linear extrapolation; works because of quasi-steady state property of benchmark model
18  currentSimEndTime = max(min(currentSimEndTime · t_opt^wct / runtime_wct, t_max), t_min);
}
```

Listing 7.1: A simple algorithm for simulation end time calibration

all simulation times $\geq t_{min}$, a linear extrapolation can be used for searching a simulation end time with the desired characteristic (line 18). The success of the search is checked in line 14, where the average execution time of all algorithms in the sample is checked to be in an interval surrounding t_{opt}^{wct}. The size of that interval can be controlled by another user-defined parameter θ ($\in [0, 1]$). Finally, a user can also configure the maximal number of search steps, so that an overly long calibration phase is avoided. To underestimate t_{end} is much less costly than to overestimate it, since execution is faster for smaller time intervals. Therefore, the algorithm is initialized with t_{min} as the best simulation end time found so far. Line 18 ensures that the simulation end time candidates are always in $[t_{min}, t_{max}]$. If a simulator exhibits an execution time $t^{wct} > t_{max}^{wct}$ for a `currentSimEndTime` $> t_{min}$, a watchdog procedure outside the main calibration algorithm stops testing the suitability of `currentSimEndTime` and resets the algorithm to try

$$\max\left(t_{min}, \frac{\text{currentSimEndTime} \cdot t_{max}^{wct}}{(1+\theta) \cdot t^{wct}}\right)$$

as a new candidate. Again relying on a simple linear interpolation, the new simulation end time should result in a wall-clock execution time for this simulator that is just below t_{max}^{wct} by the given tolerance θ (and not smaller than t_{min}). If t_{min} is initialized correctly, i.e., not too small for the benchmark model configurations at hand, so that their quasi-steady-state property holds, the algorithm ensures an end time t_{end} that allows a valid performance analysis ($t_{end} \geq t_{min}$), while the average wall clock time performance of the algorithms should be close to t_{opt}^{wct}.

Enhancements The simple algorithm can be enhanced in many ways. For example, it should be straightforward to integrate simple prediction methods, e.g., nearest neighbor (see sec. 2.3.1, p. 25), that generate 'good guesses' for an initial t_{end} by considering the solutions for rather similar configurations of the benchmark model. When discrete-event simulators are to be evaluated, one could also alter the algorithm to calibrate the overall number of events to be simulated, instead of the simulation end time. While this would be a more 'natural' metric for such algorithms, it does not work when approximative methods are included, such as τ-leaping (see sec. 1.3.1, p. 4).

7.3.3. Automated Performance Exploration with JAMES II

This section introduces the basic components that, by relying on the adaptive simulation runner (sec. 7.2) and the simulation end time calibration (sec. 7.3.2), enable efficient automated performance experiments. Related attempts to automate the collection of performance data to solve the algorithm selection problem are briefly discussed in the next section (general methods to speed up such experiments have already been discussed in ch. 3, p. 61). Then, different kinds of performance studies are identified, and it is discussed how the presented components can be combined to support such studies.

Related Work

In [317, 318], Vuduc et al. introduce a stopping criterion for searches within the algorithm space $\mathbb{A}$; a necessity in their case, since they consider empirical tuning (see sec. 2.7, p. 54). The algorithms are generated, i.e., $\mathbb{A}$ is prohibitively large. All performances are normalized to $(0, 1]$. Their stopping rule aborts the search in case $Pr[p_m \leq 1 - \epsilon] < \alpha$, where p_m denotes the maximal performance observed so far and ϵ and α are user-defined parameters. In other words, their stopping rule makes the search for a better implementation go on until the probability that the currently best observed performance (p_m) does *not* belong to the ϵ percent best performances overall is less than α. Vuduc et al. advise to not check the stopping criterion immediately after the search started — the data might be too much distorted — and it was found to overestimate the quality of the taken samples, although it still worked well in practice [319].

Vadhiyar et al. [311] make use of optimization methods that speed up their experiment setups by a factor of 10 to 300 when compared to normal parameter scanning (see sec. 3.2.2, p. 68).

Types of Experimentation Studies

Even if there is a suitable benchmark model that exhibits all properties described in section 7.3.1 (p. 147), it is still unclear how to automate the actual experimentation. What issues arise in practice, when exploring the performance of an algorithm set $\mathbb{A}$? To better support performance analysts in these tasks — which are a prerequisite for data-driven algorithm selection — three different kinds of performance experiments have been identified. They mainly differ in their objectives, i.e., the questions to be answered:

- ***Algorithm-centric (AC)***: Experiments that thoroughly explore the behavior of a single algorithm, usually carried out by developers.
  Main questions: *Where are the bottlenecks of the algorithm? In which dimensions does it scale? How does it compare to alternative approaches?*

- ***Trade-Off-centric (TC)***: Experiments that identify regions in a benchmark model's parameter space where the algorithm ranking regarding a performance measure is changing, e.g., the execution speed of algorithms $a_1, a_2 \in \mathbb{A}$. They are usually carried out by performance analysts.
  Main question: *For which parameters is algorithm a_1 outperforming algorithm a_2?*

- ***Exploration-centric (EC)***: Experiments that compare a set $A \subseteq \mathbb{A}$ of algorithms. They are usually carried carried out by performance analysts and deployers.
  Main question: *How do the available algorithms perform for certain benchmark model configurations?*

Following the categorization of empirical research on algorithms from [163, p. 216–217], algorithm-centric studies are typically conducted to support *"horse race papers"*, i.e., publications of new algorithms. Such studies shall evaluate the benefits of a new algorithm in comparison to others. In contrast, trade-off- and exploration-centric studies are performed for *"experimental analysis papers"*, i.e., comparative analyses for larger sets of algorithms. The distinction between TC and EC studies is their focus on either the specific trade-off points of algorithms, which might suffice in some cases,

or on a thorough algorithm performance exploration, even in regions where changes in algorithm performance ranking are unlikely. While EC studies are generally the most exhaustive ones and also contribute to the validation of the involved algorithms, TC experiments might be executed much faster by relying on extrapolation methods for finding 'interesting' regions of the overall simulation performance space $\mathbb{P} \times \mathbb{A} \times \mathbb{R}^n$ (see sec. 7.3, p. 145). These regions are not only of particular importance for algorithm selection (as they outline the approximation form to be learned) but could also be investigated by subsequent EC experiments.

A system that aims at automating the runtime performance analysis of simulation algorithms should support all three kinds of studies. All of them can be enhanced by sophisticated methods from statistics or experimental design (see sec. 3.2, p. 66). Making the integration of such methods as easy as possible will help performance analysts and developers in setting up effective and meaningful experiments. Finally, the system should allow for a calibration of the simulation end time as discussed in section 7.3.2 (p. 150), and should be able to exploit all available resources.

Components for Automated Simulation Space Exploration

This section presents a simple yet flexible mechanism to automate simulator runtime performance evaluation in the context of JAMES II. Its major components are outlined in figure 7.17. The central entity is `ISimSpaceExplorer`, an interface that extends the typical interface of an experiment steerer (see sec. 7.1, p. 132).

The interface is implemented by the `AbstractSimSpaceExplorer` class, which is based on the Strategy pattern [98, p. 315] and shall support all explorations of $\mathbb{P}$ and $\mathbb{A}$. It handles the interplay between the (optional) calibration of the simulation end time by an implementation of `IModelCalibrator` on one hand, and an algorithm to pick benchmark model configurations on the other hand. `AbstractSimSpaceExplorer` distinguishes three phases: `START_CALIBRATION`, `CALIBRATION`, and `EXPLORATION`:

1. The first phase is needed to initialize the calibrator and to select the benchmark model configuration of interest. The element is selected by an exploration algorithm to be implemented by a sub-class, so the abstract method `newModelSetup()` is called.

2. In the `CALIBRATION` phase, the simulation explorer queries the calibrator and then propagates the potential simulation end time t'_{end}, as well as the sample algorithm, to the experimentation layer of JAMES II (sec. 7.1, p. 131). It continues to work as a proxy for the calibrator, until `IModelCalibrator.done()` is true. Then, the best t'_{end} is retrieved from the calibrator and the `EXPLORATION` phase begins. If no calibrator is set, the default simulation end time will be used and this phase of the algorithm is skipped.

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

Plug-in Types & Auxiliary Components Both calibration and exploration algorithms, i.e., implementations of `IModelCalibrator` and `ISimSpaceExplorer`, are likely to incorporate methods from experiment design or additional heuristics. They are hence defined as plug-in types. Again, the `SelectionTreeSet` already discussed in section 7.2.1 (p. 136) can be used to automatically construct the set $\mathbb{A}_x$ of eligible selection trees. Alternatively, the runtime configurations to consider can also be set manually. The model's parameter space is defined by a set of `BaseVariable` instances, another basic JAMES II class (see fig. 7.17). Finally, the `UpdateGenerator` implements the automatic generation of updates for key simulation parameters within the (deeply nested) parameter blocks. It is required for the automatic calibration of the simulation end time.

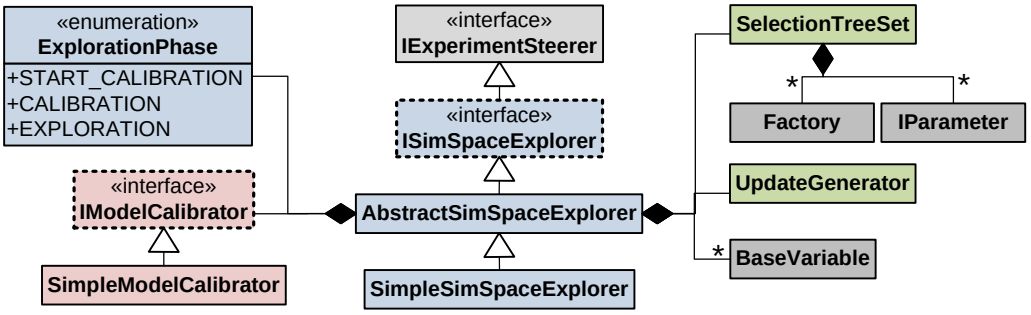

Figure 7.17.: The main components for simulation space exploration in JAMES II: the simulation space explorer (blue), the calibrator (red), and the parameter block management (green). Most are linked with fundamental JAMES II entities (grey). Plug-in types are marked by bold dotted borders.

Integration of the Adaptive Simulation Runner It is possible to harness the adaptive simulation runner (sec. 7.2, p. 134) for trade-off- and algorithm-centric experiments. This may reduce the computational efforts when testing a single benchmark model configuration. The better an algorithm performs, the more often it is chosen for execution. This reduces the number of required replications. For example, when investigating a set of 30 algorithms, instead of replicating the execution of each algorithm 20 times ($30 \cdot 20 = 600$) in order to rule out any stochastic effects of the hardware and so on (see sec. 3.2.1, p. 66), the adaptive simulation runner could choose the most promising algorithms on its own and just executes, for example, 300 replications. The number of required replications would be cut by 50% and there will still be *more* than 20 replications, i.e., *more* experimental data, for all algorithms that perform relatively well — on the expense of the algorithms that perform worse. This way of speeding up performance experiments is further detailed in section 7.3.4 (p. 157).

Performance Database and Analysis The experimental data that is generated by the performance exploration system needs to be stored for later analysis. This is done by the performance recorder described in section 5.2 (p. 113). The recorded data can then be analyzed with the SPDM (ch. 6, p. 115).

Configuring the Simulation Space Explorer for AC/TC/EC studies Figure 7.18 (p. 156) shows how to combine the components for the different kinds of algorithm performance studies (see sec. 7.3.3, p. 153). When evaluating the performance of a new algorithm (left schema), an algorithm-centric study can simply re-use suitable benchmark model configurations from the database (PerfDB, fig. 7.18). By taking advantage of the adaptive simulation runner, the number of required replications can be greatly reduced by letting it decide between the new algorithm and the fastest known algorithm for this problem instance, identified by the performance database. Having only two options will make a multi-armed bandit policy converge quickly. This idea is further pursued in section 7.3.

If only the trade-off points between two algorithms shall be found (center schema), the exploration algorithm should take algorithm performance directly into account. This helps to find regions where tipping points are likely. Since these regions are usually unknown before, the calibration mechanism should be applied to new benchmark model configurations. Again, the adaptive simulation runner can be used to reduce the number of required replications.

A thorough exploration of all available algorithms (right schema) also includes to explore the variance of each algorithm throughout the benchmark model parameter space. Therefore, using the adaptive simulation runner is not advisable for this setup; it may execute badly performing algorithms less often per problem instance, and hence might not be able to attain reliable estimates for their performance variance. Similar to the setup for trade-off-centric studies, obtained results are fed back to the exploration algorithm, so that it can decide upon new problem instances to be investigated.

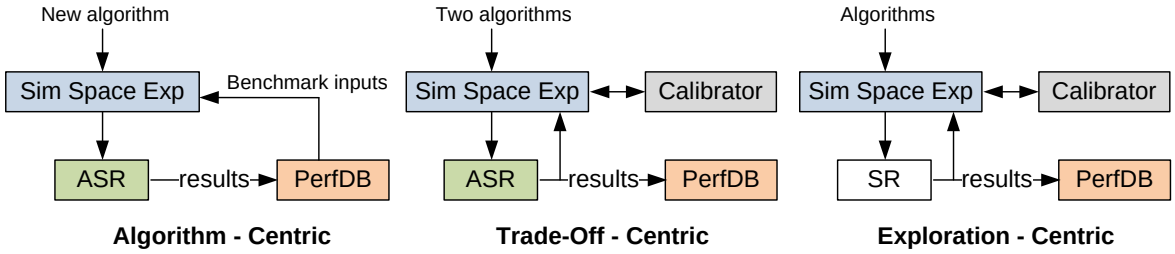

Figure 7.18.: Different setups for automatic simulation space exploration. The adaptive simulation runner (ASR) and the calibrator are not always necessary. For example, exploration-centric experiments should use another simulation runner (SR) instead of the adaptive one.

Exploration algorithms for TC and EC studies could be inspired by similar problems in experiment design (see sec. 3.2.2, p. 68).

7.3.4. Automatic Experimentation for Standard Tasks

A successful implementation of automatic algorithm selection can simplify many issues in algorithm development. As Brewer points out, *"[a]n important benefit of automatic selection is the tremendous simplification of the implementations that arose from the ability to ignore painful parts of the input range"* [25, p. 90].

On the other hand, algorithm selection also *complicates* matters for simulation developers, as already discussed for the different use cases of the SASF (see sec. 4.1, p. 79). This is because it is not sufficient anymore to enhance a *single* algorithm's performance on a *single* problem: a *range* of problems has to be considered, and it has to be checked if experimenters actually benefit from a code change. Only *peak performance* is of interest, i.e., the performance of the best-performing algorithm in $\mathbb{A}$, whenever simulation algorithm selection is applied successfully.

Hence, a successful algorithm selection mechanism changes the *objectives* of algorithm development to some extent, as Leyton-Brown et al. notice: *"Once we have decided to solve the algorithm selection problem by selecting among existing algorithms using a portfolio approach, it makes sense to reexamine the way we design and evaluate algorithms. Since the purpose of designing a new algorithm is to reduce the time that it will take to solve problems, designers should aim to produce new algorithms that complement an existing portfolio rather than seeking to make it obsolete"* [205, p. 40].

Apart from a shift in the mindset of developers, this changing perspective on algorithm performance also motivates new tools; tools that are specifically designed to support the development of simulation algorithms for flexible and extensible simulation systems like JAMES II. One important issue such tools should tackle is the evaluation of simulation algorithms. As described above, testing the algorithm in isolation does not suffice anymore — its performance has to be set into relation to the *other* algorithms. Such evaluations help to focus on the important regions in the problem space, an advantage also mentioned by Pfahringer et al. [257]. Leyton-Brown et al. ask a similar question: *"[...] if we reject the notion of winner-take-all algorithm evaluation, how ought novel algorithms to be evaluated?"* [205, p. 33]

The simulation space explorer presented in section 7.3.3 can be extended to facilitate this task by automatically configuring and executing meaningful performance experiments. The experimental results shall give developers a quick feedback on the impact of the code change they wish to explore. This may help to avoid unnecessary or bad optimizations. As William Wulf puts it: *"More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity"* [335, p. 796].

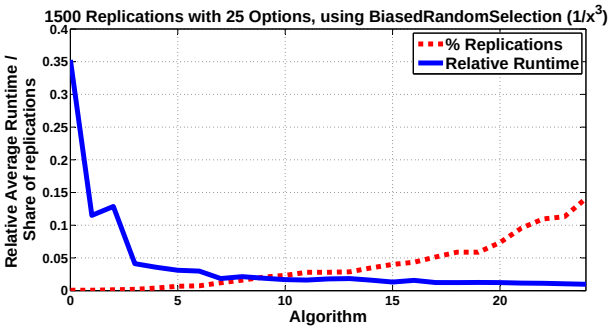

Figure 7.19.: Replication share and execution times when using a biased random selection policy for comparing SSA simulators (from [67]). After initialization, the policy randomly selects a given algorithm with a (normalized) probability $\frac{1}{\hat{R}_i^3}$, where $\hat{R}_i$ is the average execution time of algorithm $a_i \in \mathbb{A}$ so far.

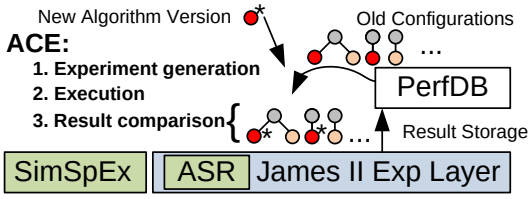

Figure 7.20.: Basic idea of the algorithmic change evaluator (ACE). A change to an existing algorithm (red node) is evaluated by considering the selection trees it is involved in and letting them compete against other best-performing selection trees for certain problems in the performance database. It relies on the simulation space explorer (SimSpEx) and the adaptive simulation runner (ASR).

Related Work

Similar support for automated experiments is required whenever a selection mapping needs to be adapted to the experimenter's hardware. This has already been implemented by several ASP solutions, e.g., an automated calibration to the hardware is realized in [318, 338]. There are also several libraries that already make use of such mechanisms and hence provide *portable performance*, e.g., the MPI [318]. Brewer [25] illustrates this requirement by comparing performance across different platforms. He notices several differences due to the differing relations between computational cost and communication overhead. In contrast, the following mechanism shall automatically check on the impact of code changes, not on hardware changes.

Implementation

Realizing an evaluation mechanism for algorithm changes — an *algorithmic change evaluator (ACE)* — just requires to recombine some of the mechanisms introduced in this chapter. The adaptive simulation runner outlined in section 7.2 (p. 134), with its ability to learn at runtime which algorithm performs best, can be easily exploited for speeding up algorithm comparisons. Here, the basic idea is that it is much more interesting to know which algorithm is the best choice for a given simulation problem, than it is to know which one is the worst choice. Consequently, simulation replication for performance comparison should be biased towards faster algorithms, thereby accepting that the performance ranking of algorithms that perform worse is not investigated as thoroughly. An example for this kind of enhanced algorithm comparison is given in figure 7.19, where several SSA variants have been compared on a CCS benchmark model configuration (see sec. 7.3.1, p. 149). The execution is replicated 1500 times, i.e., with ≈ 50 replications per algorithm. This results in many replications for the faster algorithms (> 150 for the fastest), and very few for slow algorithms (just one for the slowest). In other words, we get *more* substantial data on the performance of *interesting* algorithms in *less* time. The approach works best when the performance variance due to stochastic effects (see sec. 3.2.1, p. 66) is relatively small in relation to the overall performance.

However, it is important to notice that an efficient algorithm comparison demands for replication policies that differ from those for a good algorithm selection in two aspects: Firstly, a policy has now to ensure that each arm is played at least once, which is not necessarily the case (e.g., when using policies like ϵ-Greedy, see sec. 7.2.1, p. 136). Secondly, overall regret is not the single figure of merit anymore — instead of exploiting a single arm, it might now be advisable to throttle the exploitation and also replicate the second-best or third-best algorithm more often. The overall goal now includes

to maximize the *statistical efficiency*, i.e., the degree of certainty with which the performance ranking of the best x algorithms can be established. In future, such comparisons could be enhanced even further by including policies that also contain stopping rules, so that the algorithm comparison stops if a predefined degree of certainty is reached for the performance ranking.

The algorithmic change evaluator builds upon the versioning feature for runtime configurations, as implemented by the performance database (see sec. 5.1.1, p. 104). The workings of the mechanism are illustrated in figure 7.20 (p. 157). Whenever a developer makes a significant change to the code base, the affected algorithm can be submitted to the ACE as a new version. The ACE queries the performance database to find all benchmark model configurations whose model URI matches a given string (e.g., to restrict analysis to certain formalisms or even a single model). For each found problem, the ACE generates and executes a JAMES II experiment that compares the performance of several configurations by using the adaptive simulation runner. Which configurations to compare is up to the developer. Per default, all configurations that rely on the modified algorithm (e.g., an event queue) are reevaluated, together with the best-performing configuration that does not rely on this algorithm. This allows to assess the overall impact of the code change on peak performance, without wasting too much time on experimentation.

7.4. Summary

After outlining the experimentation layer of JAMES II (sec. 7.1, p. 131), this chapter introduced two important components: an adaptive simulation runner (sec. 7.2, p. 134) that allows to identify good selection mappings at runtime, and the simulation space explorer, a component that combines the adaptive simulation runner with simulation end time calibration (sec. 7.3.2, p. 150), in order to speed up and automate the experiments to collect performance data. The applicability of the adaptive simulation runner can be enhanced by restricting its choice to simulation algorithm portfolios (sec. 7.2.2, p. 140). Finally, a simple component to set up automatic performance experiments that support simulation algorithm development was described (sec. 7.3.4, p. 156).

Most of the components introduced in this chapter have already been used for performance studies; they will be evaluated in the case studies (ch. 9 and ch. 10). Their usage will hopefully support efforts on high-quality simulation algorithms, which — in case of stochastic simulation — will have an immediate effect on overall system performance when using the adaptive simulation runner [67].

This chapter presented many new methods — however, they should be considered the tip of the iceberg. While the application of multi-armed bandit policies for simulation replication is quite straightforward, the interrelation of portfolio construction techniques, their parameters (e.g., regarding portfolio size), and the performance of multi-armed bandit policies motivates additional investigations (as becomes apparent in the results of ch. 9, p. 177). Similarly, the simulation exploration mechanisms currently either solve problems that would otherwise prohibit any automated large-scale performance experiments, e.g., by simulation end time calibration, or they are still quite preliminary, e.g., the algorithmic change evaluator would greatly benefit from a dedicated user interface or the integration into an integrated development environment. This would make it much more accessible to developers — so far it is just a prototype. Furthermore, many more techniques discussed in chapter 3 (p. 61) could be included, e.g., from the field of meta-modeling. All this should be regarded as future work.

A crucial element to close the feedback loop is still missing. Semi-automated large-scale performance experiments can be conducted, their results can be stored (ch. 5, p. 101) and used to generate selection mappings (sec. 10, p. 197) — but these results also need to be *deployed* to the simulation system, so that it can actually perform an automatic selection of simulation algorithms. A prototypical extension of the JAMES II registry that allows to do so is presented in the next chapter, which also includes a basic evaluation of the overall effectiveness of the framework (see fig. 4.6, p. 97).

8. Automatic Simulation Algorithm Selection in JAMES II

On two occasions I have been asked,—"Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?" ... I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.

Charles Babbage, *Passages from the Life of a Philosopher*, 1864

This final chapter of the thesis' methodological part covers concrete steps towards automated simulation algorithm selection. The chapter describes a prototypical extension of the JAMES II registry, so that automated algorithm selection is supported in a way that does not affect the existing code base. It describes the management and exploitation of the performance data analyses, as depicted in figure 8.1.

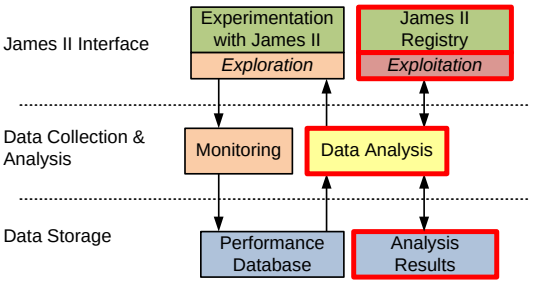

Figure 8.1.: SASF overview (see fig. 4.6, p. 97), red borders denote elements discussed in this chapter.

Apart from the adaptive simulation runner presented in section 7.2 (p. 134), which is part of the experimentation layer of JAMES II (see sec. 7.1, p. 131), it is not yet clear how automated algorithm selection can be integrated into JAMES II. The main process to be supported has already been motivated (sec. 4.1, p. 79) and outlined (sec. 4.4.2, p. 96): data collected from performance experiments (sec. 7.3, p. 145) are stored (ch. 5, p. 101) and subsequently analyzed by the SPDM, which generates selectors to solve the ASP (see ch. 6, p. 115).

The first part of this chapter describes how these selectors can be deployed to JAMES II. It introduces a prototypical extension of the JAMES II registry (see sec. 4.2.1, p. 84), which accommodates selectors and uses them if necessary. The second part of this chapter evaluates the overall effectiveness of the simulation algorithm selection framework (SASF) in generating and using selectors of acceptable quality. This is done via a fully implemented synthetic setup—i.e., by dedicated plug-ins for a model formalism and simulation algorithms—which allows to assess the potential performance benefits of using the SASF. More realistic examples are studied in the last part of the thesis (ch. 9 and ch. 10).

8.1. An Algorithm Selection Registry for JAMES II

As already described in section 4.2.1 (p. 81), the JAMES II registry is responsible for selecting the most appropriate plug-in for a given task at runtime. Application code simply calls the registry's `getFactory(...)` method. Given an abstract factory class and parameters that specify the actual task, it returns a factory to create a suitable component that complies to the requirements implied by the parameters (see p. 84).

This design allows the extension of JAMES II for automated algorithm selection in a rather straightforward manner: the registry can be replaced by a subclass that overrides `getFactory(...)` to integrate automated algorithm selection. As the method is already in broad use to request a suitable plug-in from the JAMES II registry, this is the least-intrusive way of integrating a more sophisticated algorithm selection mechanism—no further program code needs to be changed.

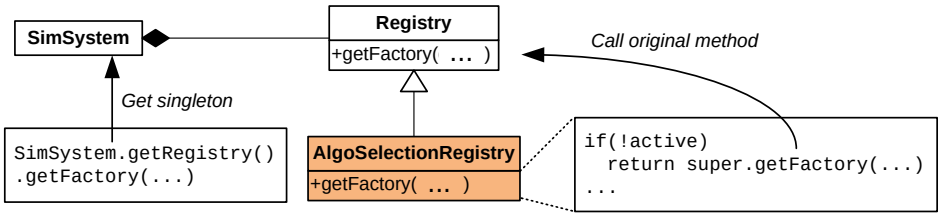

Figure 8.2.: Integration of the `AlgoSelectionRegistry` into JAMES II. Application code stills retrieves the registry singleton from `SimSystem` and thus remains unchanged. If the registry is an instance of `AlgoSelectionRegistry`, it can still be used as if it was the original JAMES II registry, by setting the `active` flag to `false`.

The problems that arise when applying algorithm selection recursively without context — as discussed in section 4.2.3 (p. 87) — are circumvented by not introducing the algorithm selection mechanisms as filter criteria (as originally planned, see [70]) but rather *complementing* the existing filtering mechanisms with *suggestions* from algorithm selectors. Filter criteria operate on a single *kind* of plug-in, e.g., event queues, whereas the selectors described in chapter 6 (p. 115) select the *whole* runtime configuration that is predicted to perform best. Furthermore, filter criteria are always applied, whereas selectors are typically confined to certain sub-spaces of the problem space $\mathbb{P}$, e.g., to certain modeling formalisms. This makes it necessary to first pick a suitable selector, which then selects the whole runtime configuration — a procedure that may allow the addition of meta-learning approaches in the future (see fig. 2.15, p. 59).

In case this new, more complex behavior of the registry is not desired, it can be switched off easily by setting a corresponding flag. If this is done, calls to `getFactory(...)` merely return the original registry's choice — no automatic selection based on problem features takes place. Figure 8.2 summarizes the basic scheme of extending JAMES II towards automatic algorithm selection.

Although the new registry class — `AlgoSelectionRegistry` — only needs to override one method of its super class, the deployment of SPDM selectors requires additional mechanisms. Selectors are intended to be generated by the SPDM before processing time (see sec. 2.6.1, p. 50) — usually even before run time, as not all users will have large enough performance databases at their disposal. Hence, the selectors need to be stored somewhere, and they need to be associated with the plug-in type they apply to (e.g., simulation algorithms). To dynamically associate additional meta-data with plug-ins and plug-in types is not supported by the JAMES II registry, it therefore has to be provided by the `AlgoSelectionRegistry`. Section 8.1.1 describes a mechanism that implements this, and also introduces the notion of a plug-in life cycle. Then, section 8.1.2 describes how the meta-data can be used to automate algorithm selection with respect to another fundamental requirement, namely the *robustness* of a system in terms of failure handling (see sec. 5.1.1, p. 108, and sec. 2.4.3, p. 36). This is possible by considering the faulty runtime configurations that have been quarantined by the adaptive simulation runner (see sec. 7.2, p. 134). Finally, section 8.1.3 details the deployment of SPDM selectors to the `AlgoSelectionRegistry`, and how they are used.

8.1.1. The Plug-in Life Cycle and the Plug-in Data Storage

Before discussing how the meta-data is stored, it has to be clear what *kind* of data needs to be stored. Besides SPDM selectors, this is currently the status of a plug-in. The *plug-in status* is a phase in the *plug-in life cycle*. Such a (prototypical) life cycle for JAMES II plug-ins is depicted as a UML state chart in figure 8.3 (p. 161).

According to the proposed life cycle, all plug-ins start out as being under development (see initial state in fig. 8.3). If its developer regards it as finished, it may be *submitted*, i.e., released for general usage. Then, some standard tests — e.g., unit tests — are carried out. If the new plug-in passes these tests, it is regarded as *tested* and can now be applied to real-world problems. After being in use

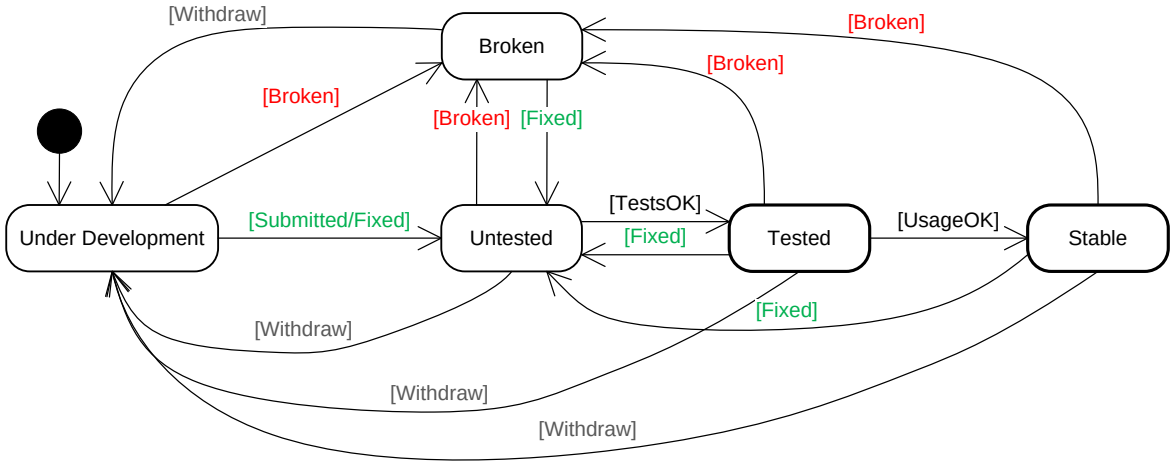

Figure 8.3.: The plug-in life cycle considered by the `AlgoSelectionRegistry`, depicted as a UML state chart (e.g., [86, p. 121 et sqq.]). It is implemented in the class `ComponentState`. The colored transition guards of each kind all go to the same state: all fixed plug-ins are regarded as `untested` (green), all plug-ins that fail are regarded as `broken` (red), and all withdrawn plug-ins are regarded as being `under development` (grey).

for some time without failures or problems, a plug-in might eventually be considered *stable*. Only tested or stable plug-ins (demarcated by bold borders in fig. 8.3) should be used by experimenters for productive simulation studies, i.e., studies where the specific simulation outcome matters.

Performance analysts may also want to use untested plug-ins; developers even need to execute plug-ins that are broken or under development (e.g., for debugging). Hence, the `AlgoSelectionRegistry` can be configured with a role-dependent *failure tolerance* of a user. The tolerance may be set to `ACCEPT_ALL`, `ACCEPT_UNTESTED`, `ACCEPT_TESTED`, or `ACCEPT_STABLE`. The given order of tolerance levels refers to their strictness: if the tolerance level is set to `ACCEPT_TESTED`, this also includes stable plug-ins, but not untested ones.

The state of a plug-in is associated with its concrete factory, which is already managed by the JAMES II registry. SPDM selectors, on the other hand, should be associated with plug-in *types*, as they shall serve as tie-breakers in case more than one plug-in is eligible (see sec. 4.2.1, p. 85). Two additional classes are introduced, `FactoryRuntimeData` and `AbstractFactoryRuntimeData`, to store these data for factories and abstract factories, respectively. `FactoryRuntimeData` contains the current state of a plug-in (see fig. 8.3), as well as a variable to count how often the given plug-in was used successfully, i.e., without a failure. This additional information may help to decide at which point a plug-in can be regarded as stable. The counter is reset in case the plug-in is reported broken. So far, the decision to regard a plug-in as stable has to be made manually. `AbstractFactoryRuntimeData` merely contains a `SelectionEnsemble`, a class that is detailed in section 8.1.3.

A new interface, `IFactoryRuntimeDataStorage`, is defined for components that manage the registry's meta-data. It allows to associate instances of `FactoryRuntimeData` and `AbstractFactoryRuntimeData` with factories. While this would be a typical situation to introduce a new plug-in type for the storage of registry meta-data, the `AlgoSelectionRegistry` is just a prototype — only a simple file-based implementation of `IFactoryRuntimeDataStorage` currently exists. Alternative storage schemes, e.g., for synchronizing with a central repository to get meta-data updates from other users, can still be implemented easily (and should then be included as plug-ins of a new plug-in type). Furthermore, the meta-data storage may be enhanced so that plug-ins can store custom data as well, such as usage statistics or configuration details.

8.1.2. Automated Failure Detection

If a runtime configuration of JAMES II fails, it is often hard to tell *which* component caused the problem. Imagine a runtime configuration that contains a simulation algorithm and an event queue as subordinate component: if the configuration crashes, which of the components caused it? Did the event queue mix up events, or did the simulator access the queue in a wrong manner? While the adaptive simulation runner is able to detect such problems (given they lead to a Java exception), it only quarantines the runtime configuration in question until the simulation problem has been solved, i.e., its multi-armed bandit policy finished replicating.

This crucial information on algorithm performance should not be wasted:[1] if JAMES II is configured to use the `AlgoSelectionRegistry`, the adaptive simulation runner calls an additional method, `reportFailure(...)`, which passes an instance of the class `FailureDescription` to the registry. In case automatic failure detection is enabled, the registry propagates this report to the `FailureDetector`, which returns a `FailureReport`.

The `FailureDescription` contains the exception that describes the failure, a description of the simulation problem, and the set of factories that were involved in the simulation. The `FailureReport`, returned by the `FailureDetector` after it has processed a `FailureDescription`, is either empty[2] or points to the factory representing the plug-in that has been detected broken. If a plug-in has been detected broken, the `AlgoSelectionFactory` will change its life cycle status to `Broken` (see fig. 8.3, p. 161). The plug-in will not be selected again for any simulation task, unless the user's fault tolerance is set to `ACCEPT_ALL`. The descriptions of all failures caused by the broken plug-in is also included in the `FailureReport`. Such kind of 'evidence' may come in handy for debugging the faulty component.

A component like `FailureDetector` is nothing unusual in adaptive software systems; e.g., Karsai et al. propose a rather similar component named 'fault diagnostics system' [176, p. 34–35]. The difficulty of detecting failures in JAMES II plug-ins is that of limited data: given a *combination* of components that failed — i.e., a selection tree (see def. 5.1.1, p. 105) — it is not clear *which* of the involved plug-ins is actually broken. Declaring them all broken, just to be be on the safe side, is no option here: if there is one plug-in that cannot be replaced by an alternative plug-in, the simulation problem cannot be processed anymore — even though the irreplaceable plug-in is fine and the error is somewhere else. It is hence necessary to carefully interpret the failure descriptions issued to the `AlgoSelectionRegistry`, and to only declare a plug-in broken after some more evidence suggests it. A simple algorithm to do so is presented in the following.

Identifying Single-Factory Failures

The failure detection algorithm is focused on detecting single plug-ins that fail. If a `FailureDescription` arrives, each involved plug-in will be added to the list of *suspects*, if it is not already contained in this list. Along with each suspect, the failure detector stores a list of that suspect's 'co-defendants', i.e., plug-ins that so far have *always* been involved in failures that also involved the suspect plug-in. If a plug-in is contained in a `FailureDescription`, its list of co-defendants will be checked: if a co-defendant is *not* involved in the current failure, it will be removed from the list.

The rationale behind this rule is Occam's razor [333, p. 151]: if the co-defendant plug-in has been involved in some failures, but not in others, while the suspect plug-in was involved in *all* of these failures, the simpler hypothesis would suggest that the suspect — and not the co-defendant — is broken. Hence, the co-defendant is removed from the co-defendant list of the suspect plug-in. If a suspect plug-in has no co-defendants left, this means that there is a set of failures with only a single commonality: the presence of the suspect plug-in. In this case, the `FailureReport` generated by the `FailureDetector` points to the given plug-in and contains the description of all failures that led to

[1] This could be regarded as a first step toward a more *robust* software system, as discussed in the context of autonomous computing (sec. 2.4.3, p. 36).

[2] This may happen in case results are inconclusive, as in the above example with event queue and simulator.

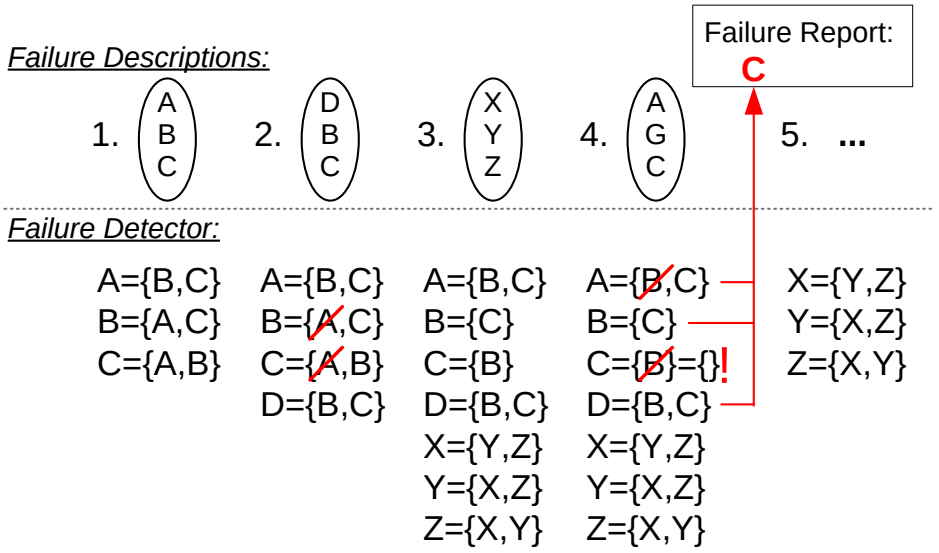

Figure 8.4.: Basic idea of automatic failure detection in JAMES II. The failure descriptions contain the plug-ins involved $(A, B, \ldots, Z)$, the sets below the dotted line denote the co-defendants of each suspected plug-in. After considering four failure descriptions, plug-in C has no co-defendants left and is hence convicted to be broken. Consequently, all algorithms that have C in their list of co-defendants are rehabilitated and removed from the list of suspects. No new algorithms from this failure description (e.g., G) are registered. If one of them is also faulty, it might still be convicted, based on further failure descriptions.

its conviction, i.e., the *evidence* with which it was detected to be broken.[3] Finally, all suspect plug-ins that contain the convicted plug-in in their list of co-defendants are rehabilitated, i.e., they are removed from the list of current suspects. The basic idea is illustrated in figure 8.4.

This approach is able to detect a plug-in that causes failures when it is combined with at least two *different* components for each plug-in type; it is not able to do so for *combinations* of algorithms. For example, if event queue E and simulator S always fail when *combined* — but not in combination with other event queues or simulators — this will go undetected. The algorithm may operate in either strict or non-strict mode: in non-strict mode, a plug-in gets rehabilitated if it has once been applied successfully. In strict mode, suspect plug-ins remain suspects. An additional parameter allows to control how many failures a plug-in has to be involved in before it can be convicted of being broken. Note that a conviction is *not* necessarily true; just the evidence in form of failure descriptions suggests that this might be the case. This kind of basic failure detection should merely prevent frequent crashes, and suggests where the error *could* be located.

8.1.3. Integration of Selection Mappings

Several steps are necessary to fully integrate the products of the simulation algorithm selection framework with the rest of JAMES II. Firstly, developers need to exert some control over the way automatic algorithm selection is carried out by the `AlgoSelectionRegistry`. This is implemented by introducing some dedicated Java annotations. Secondly, the `AlgoSelectionRegistry` has to manage *multiple* SPDM selectors per abstract factory, e.g., one to solve the ASP for chemical reaction networks and another one to solve the ASP for a parallel and distributed discrete-event simulation. Since both selectors choose from a set of processor factories, i.e., simulators, there has to be an additional mechanism to select one of them that suits the given simulation problem. Finally, the `AlgoSelectionRegistry` needs to carry out the actual selection procedure. This is discussed in the last subsection.

[3] Failure detection stops analyzing the current failure description in case of a conviction, so no explicit tie-breaking is necessary.

Annotations for Algorithm Selection

JAMES II provides various plug-in types (see fig. 1.2, p. 3); not all of them shall be selected automatically. For many plug-in types, e.g., model editors or user interface components, an automatic algorithm selection does not seem desirable. Therefore, the `AlgoSelectionRegistry` only supports automated selection for plug-in types that have been associated with a custom *Java annotation* named `AlgorithmSelection`, to decorate a plug-in type's abstract factory. Java annotations are simple data structures that can be associated with Java entities (such as interfaces, classes, or methods) to convey additional meta-information. For example, the built-in annotation `@Override` can be used to mark methods that are intended to override existing methods, thereby allowing the compiler to check if they really do — if not, there might be a software bug.

Since the `AlgorithmSelection` annotation needs to be available to all plug-in types, it is located in the JAMES II core (package `james.core.algoselect`). To let the `AlgoSelectionRegistry` automatically select algorithms of a given plug-in type (e.g., simulators), its abstract factory has to be annotated as follows:

```
@AlgorithmSelection(SelectionType.TREE)
public class AbstractProcessorFactory extends AbstractFactory<ProcessorFactory> {
...
}
```

The additional parameter `SelectionType.TREE` merely specifies that upon selecting a suitable algorithm for this plug-in type, the `AlgoSelectionRegistry` is allowed to pre-select the whole selection tree, i.e., the whole runtime configuration (see sec. 5.1.1, p. 104), in a single step. It is the only supported mode of algorithm selection so far. In a tree-like selection, arbitrary sub-algorithms may be selected as part of a runtime configuration, regardless of the annotation their own abstract factory has. Adding a similar annotation to another abstract factory, e.g., for event queues, would just mean that *individual* selectors for selecting event queues can be deployed to the registry and will be used. Since the focus is on *simulation* algorithm selection, the above annotation for simulation algorithms suffices so far. Nevertheless, the enhanced registry in principle supports automatic selection of arbitrary algorithms.

Note that the `AlgorithmSelection` annotation is *only* required for abstract factories, i.e., it is in the responsibility of the deployer. Individual developers may provide new plug-ins for certain plug-in types, but to enable automatic selection for *all* plug-ins of a plug-in type merely requires to annotate a single class: the abstract factory. The annotations are checked by the `AlgoSelectionRegistry` at runtime.

Two additional annotations, `Broken` and `UnderDevelopment`, shall support developers in adjusting the status of plug-ins they work on. Both annotations are checked at start-up by the plug-in data storage (see sec. 8.1.1, p. 160), which then sets the status of the annotated plug-in accordingly (see fig. 8.3, p. 161). This allows developers (and deployers) to easily configure the `AlgoSelectionRegistry` to avoid the plug-in in question. Without such annotations and meta-data on the current plug-in status, the only alternative would be to remove the problematic plug-in from the registry altogether, e.g., by adjusting its plug-in definition file. This, on the other hand, will prevent the selection of the plug-in in *any* case and would make it necessary that developers and experimenters use *distinct* versions of the simulation system. These problems are rendered obsolete by explicitly annotating plug-ins that should currently be avoided by experimenters; this facilitates deployment and maintenance.

Selector Managers and Selector Ensembles

As already mentioned, the `AlgoSelectionRegistry` has to be able to manage *multiple* SPDM selectors per abstract factory, e.g., selectors that rely on features from different modeling formalisms. All selectors that are defined for a certain plug-in type — e.g., all simulation algorithm selectors — are managed by a single object of type `SelectorEnsemble`. The `SelectorEnsemble` can be regarded as

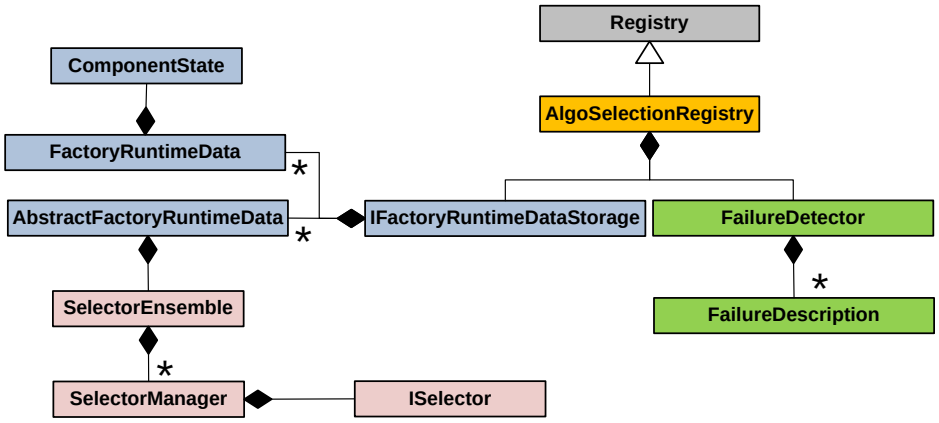

Figure 8.5.: Overall structure of `AlgoSelectionRegistry`: the prototypical new registry class (orange) extends the original JAMES II registry (grey), contains a sub-system for failure detection (green), and stores meta-data on plug-ins and plug-in types (blue). SPDM selectors (see fig. 6.12, p. 128), wrapped by `SelectorManager` objects and managed by instances of `SelectorEnsemble` (red), are part of the meta-data.

some kind of rather complex meta-data, representing knowledge on algorithm performance for the given plug-in type.

While a `SelectorEnsemble` may also employ meta-learning to improve the selection of suitable selectors (see fig. 2.15, p. 59), at the moment it merely serves as a container for `SelectorManager` instances. A `SelectorManager` encapsulates an SPDM selector and hence provides the integration logic to mediate between the simulation algorithm selection framework and the original JAMES II registry. The `SelectorManager` decides whether its selector is able to select an algorithm for a given simulation problem, which is characterized by a `ParameterBlock` (see sec. 4.2.1, p. 84). If so, it extracts the problem features and applies the selector to the problem. This level of generality allows to define different kinds of integration logic for selectors, which are completely wrapped and hence transparent to the `AlgoSelectionRegistry`. Figure 8.5 gives an overview of all relevant classes and interfaces, and how they are interrelated.

The Selection Procedure

So far, only a simple `SelectorManager` implementation is provided. It supports simulation algorithm selectors defined for a certain modeling formalism, i.e., the applicability of the selector to a problem is checked by considering the interfaces of the model that is passed along within the parameter block. Additionally, the current selector manager implementation queries the registry to find all feature extractors that are subclasses of `ModelFeatureExtractorFactory`, a sub-class of the base factory for feature extraction (see sec. 5.1.1, p. 106). It specifies an additional method to extract features from real-world parameters instead of the performance database. Future versions may include infrastructure feature extractors, e.g., to check available network resources, or selectors that are not applicable to all kinds of models from a modeling formalism. All this can be easily implemented on top of the current data structures.

To avoid the problem of recursively selecting algorithms without having access to the context in which they shall be used (see sec. 4.2.3, p. 87), the `AlgoSelectionRegistry` applies a suitable SPDM selector just *once*. The selector picks the runtime configuration it deems best, given the features extracted from the given simulation problem. It does so by sorting all runtime configurations (as discussed in sec. 6.2.1, p. 119) and picking the first configuration where all factories in the selection tree are available and have a sufficient status — e.g., no required plug-in is broken (see fig. 8.3, p. 161). The selected runtime configuration, more precisely its selection tree, is transformed by the `SelectorManager` into a `ParameterBlock` that configures JAMES II to use this runtime configura-

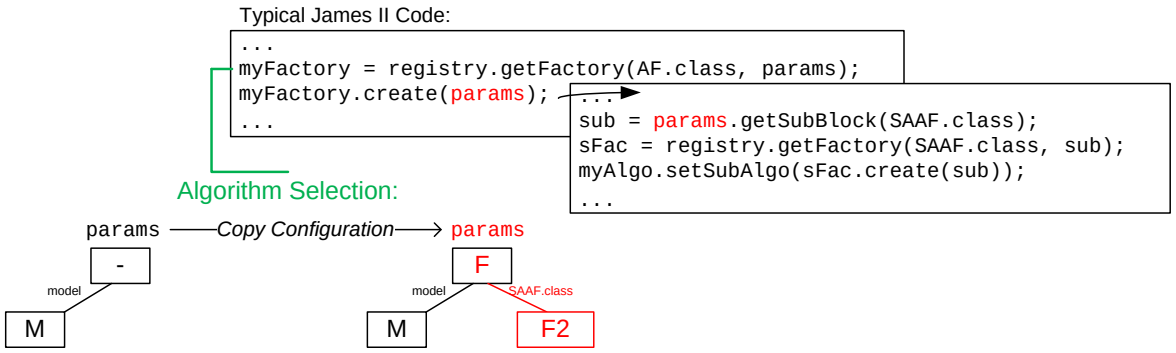

Figure 8.6.: Configuring JAMES II with a single call to the `AlgoSelectionRegistry`. Since parameter blocks are highly nested structures and get partly passed from a fractory to the factories of its sub-algorithms, it is sufficient to call algorithm selection once and let it define the whole JAMES II configuration by augmenting the given parameter block accordingly. In the given (simplified) example, the original parameter block (black) only contains a description of the simulation problem (model M). Algorithm selection may now augment this parameter block — since Java uses call-by-reference, the changes made to the parameter block (red) will then be passed to `myFactory.create(...)`, where they cause the selection of a specific sub-algorithm ($F2$). As shown in figure 8.7, the algorithm selection mechanism calls the registry again with the augmented parameter block, so that the currently desired algorithm can be selected automatically as well (here, F). Only the abstract factories (AF and $SAAF$) need to be known at compile time. The parameter block nesting can be arbitrarily deep.

tion, including *all subordinate* components. The basic approach is depicted in figure 8.6.

The advantages of this approach are that algorithm selection — including feature extraction — only has to be done *once*, and its results are transparent to the rest of the system: it does not matter whether an SPDM selector has specified the resulting parameter block or a user has done so manually.

However, this approach is prone to an issue that arises when using such up-front algorithm selection in a simulation system as flexible as JAMES II: there is still no guarantee that the selected setup is able to cope with the unknown problem, i.e., that the selected plug-ins are indeed applicable. This *problem of premature decision* can be avoided by keeping the selectors up to date. Nevertheless, it has to be taken into account by the new implementation of the `getFactory(...)` method. Figure 8.7 shows a flow chart that illustrates how `getFactory(...)` is implemented in the `AlgoSelectionRegistry`.

Automatic algorithm selection is only carried out in case no specific algorithm (i.e., factory) has been prescribed yet. Furthermore, all requirements for automated algorithm selection — annotated abstract factory, available selector ensemble, eligible selector — have to be fulfilled. If this is not the case, the `AlgoSelectionRegistry` falls back to a behavior similar to that of the original JAMES II registry. It is just complemented by an additional check of the plug-in meta-data, regarding the plug-in status and the user's failure tolerance. In case of automated algorithm selection, the chosen SPDM selector selects a certain configuration that is transformed into a `ParameterBlock` instance by its `SelectorManager` (see fig. 6.5, p. 119). Afterwards, the `getFactory(...)` method merely copies this parameter block into the one that was passed as a parameter (see fig. 8.6) and then calls itself again. The updated parameter block now defines the JAMES II configuration that was selected automatically, so all subsequent calls to the `AlgoSelectionRegistry` should follow the green path in figure 8.7; it should be able to select all factories prescribed by the new parameter block.

8.2. Testing the Effectiveness of the Overall Approach

The last chapters have introduced several sub-systems of the simulation algorithm selection framework. Before they are assessed in real-world scenarios, it seems mandatory to assess the effectiveness of the *overall* procedure, namely the execution of a performance study, the generation of selectors, and

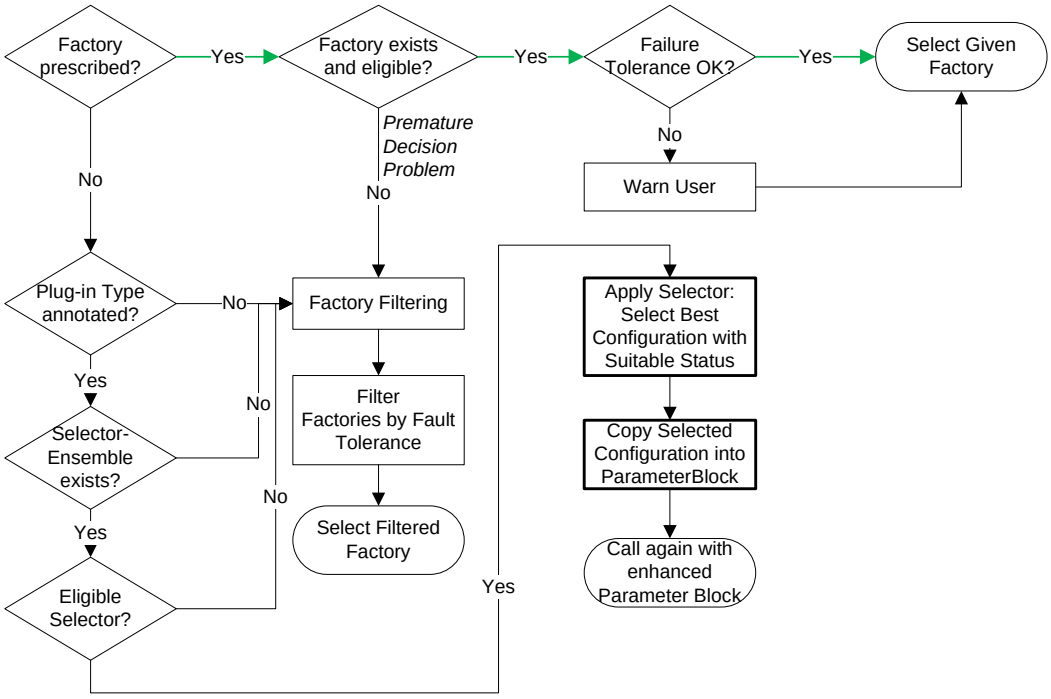

Figure 8.7.: Flow chart of the selection procedure in `AlgoSelectionRegistry`. Bold borders mark the two actions that relate to automatic algorithm selection. Afterwards, `getFactory(...)` is called recursively, now with an augmented parameter block that prescribes which factory to be used. Ideally, subsequent calls to `getFactory(...)` follow the green arrows, i.e., the selected algorithms are able to cope with the given problem.

their performance after deployment to the `AlgoSelectionRegistry`. To get a reliable estimate on the effectiveness that can be expected from the proposed ASP solution approach, the problem to be solved should be well understood and easy. In other words, the best selection mapping S^* should be known and it should be attainable, at least in theory, by considering the extracted model features.

Furthermore, all model features should influence simulator performance in a straightforward manner. While the problem of feature selection (def. 2.1.3, p. 15) is also challenging, it has not been addressed so far. More complex relationships between model features and simulator performance may be discovered by additional machine learning schemes that can also be integrated into the SPDM (see sec. 6.2.1, p. 121). Section 8.2.1 presents a test setup that complies with the above requirements. The experimental results are presented and analyzed in section 8.2.2.

8.2.1. Test Setup

Since the goal of the test setup is to check the overall effectiveness under the most simple circumstances, a synthetic scenario is preferred over a realistic one — not even the most simple model formalisms and their simulation algorithms in JAMES II are as simple and well-understood as the basic benchmark components specifically designed for this purpose. The `BogusModelFormalism` defines models which are essentially *name* $\rightarrow$ *value* mappings. Each *name* corresponds to a feature type, and its integer *value* gives the value of this feature — feature extraction is therefore trivial. Three different simulation algorithms are applicable to such models: the bogus simulators A, B, and C. All are sub-classes of the same bogus simulator and merely define on which of the three model features they rely. The model features are named after the simulation algorithm they are associated with: A, B, and C. The simulators use their corresponding feature as a factor in a simple `for`-loop, which generates

synthetic load. Hence, each simulator's execution time — the performance metric targeted here — linearly depends on its model feature and is independent of all other model features.

This scenario ensures that no simulation algorithm dominates the others. The simulation algorithm selection should hence be able to find a selector that is adaptive-effective (def. 2.1.6, p. 18): the best selection mapping should outperform *any* constant selection mapping that always picks the same simulator. An optimal selection mapping, S^*, can be defined as:

$$S^*((a,b,c)) = \begin{cases} A & a = \min(\{a,b,c\}) \\ B & b = \min(\{a,b,c\}) \\ C & c = \min(\{a,b,c\}) \end{cases} \tag{8.1}$$

where $(a,b,c) \in \mathbb{F}$ are the problem features and $A,B,C \in \mathbb{A}$ are the simulators to be selected.[4] Note that there is in fact a *set* of optimal selection mappings, since it does not matter which of the corresponding algorithms to choose in case there is more than one minimal feature. Besides that, the structure of the problem defines all algorithms to perform equally well overall. Executing each of them on the whole set of models with features $(a,b,c) \in [1,10] \times [1,10] \times [1,10]$, for example, would result in all algorithms having a *similar* overall execution time. They just outperform each other *within* the well-balanced problem space. This implies that the constant selection mappings in $\mathbb{S}_C$ (see sec. 2.1.2, p. 16) perform similarly as well, given that future simulation problem features are equally distributed over the feature space $\mathbb{F}$.

Overall Procedure

The mechanisms introduced in the last chapters should now be judged by their overall result. The ASP solution process under scrutiny consists of several steps, which are quite similar to those of other algorithm selection approaches (see sec. 4.4.1, p. 95):

1. **Performance Experiment**: The performance database (ch. 5, p. 101) is filled by using the calibration mechanism (sec. 7.3.2, p. 150) in a simple simulation space exploration setup (sec. 7.3.3, p. 154): a number of benchmark models configurations is chosen at random, where each feature value is picked from the interval $[1, p_{max}]$, p_{max} being an integer parameter to control the size of the sample space. The size of the sample, i.e., the number of benchmark model configurations, is denoted by s. Each algorithm execution is replicated r_a times. The adaptive simulation runner (sec. 7.2, p. 134) is not applied, as there are just three algorithms to deal with. For calibration, $t^{wct}_{opt} = 5\ s$, $t^{wct}_{max} = 10\ s$, and $t^{sim}_{min} = 2\ s$ were chosen. Since the calibration mechanism is used for exploration, i.e., the simulation end time varies across simulation problems, the performance of an algorithm is recorded as *throughput*, i.e., simulation end time divided by wall-clock time. This is necessary to adjust the execution time by the simulation end time (the benchmark model is in quasi-steady state, see sec. 7.3.1, p. 147). Another option would be to include the simulation end time of a problem as additional feature.

2. **Feature Extraction**: Feature extraction is done with a custom feature extractor (sec. 5.1.1, p. 106), which merely reads out the $name \rightarrow value$ map of a benchmark model.

3. **Selector Generation**: Selector generation and evaluation is done as described in chapter 6 (p. 115).

4. **Selector Deployment**: Finally, selectors are generated from *all* available performance data and deployed to the `AlgoSelectionRegistry`. Their performance is measured on two sets: first, a *new* sample set is drawn from the same problem space (characterized by p_{max}). The performance of the selectors reflects the case where the benchmark model characteristics accurately characterize the features of future simulation problems. Secondly, a *different* sample space is

[4] As there is only one user criterion to be considered by S^*, it has been omitted for clarity (see sec. 2.1, p. 13).

used to draw a new sample. It reflects situations in which selectors have to generalize adequately, as they have not been trained to select algorithms for all problems they are confronted with. Note that the latter case is still rather optimistic, since simulator performance depends *linearly* on the features and is hence very easy to extrapolate. The overall test setup can be regarded as testing with a test set and a validation set, as suggested by Hastie et al. in [122] (see sec. 6.2.2, p. 124).

Parameter Setup

The overall procedure is tested with different parameters. The parameter p_{max}, controlling the size of the problem space that is explored, is chosen from the set $\{10, 20, 30\}$. Therefore, the overall feature space $\mathbb{F} = [1, p_{max}] \times [1, p_{max}] \times [1, p_{max}]$ contains between $10^3 = 1.000$ and $30^3 = 27.000$ elements. As the models are defined so that their features completely characterize their computational load, the problem space $\mathbb{P}$ has the same size. There are three algorithms ($|\mathbb{A}| = 3$), so the overall simulation space $\mathbb{P} \times \mathbb{A} \times \mathbb{R}$ is characterized by a real-valued performance function p. Its domain contains between $3 \cdot 1.000 = 3.000$ and $3 \cdot 27.000 = 81.000$ elements. For each problem $x \in \mathbb{P}$ and for each algorithm $a \in \mathbb{A}$, p refers to the *throughput* of a when applied to x, i.e., simulation time divided by wall-clock time. The shape of this function is explored by the simulation space explorer in the first step.

The exploration of p is carried out by drawing random elements from $\mathbb{P}$; the sample size s is chosen from $\{10, 20, 30\}$. Since each of the three algorithms in $\mathbb{A}$ is applied at least once to each problem, this results in 30 to 90 performance tuples that are handed over to the SPDM for selector generation. To control the impact of stochastic noise during simulation space exploration, the number of replications per algorithm, r_a, is varied between 1 and 3. Replicating only once yields less reliable performance data, and should hence lead to selectors that perform less well. Selectors generated from data averaged over three replications, i.e., with $r_a = 3$, should perform better.[5] Similarly, larger sample sizes s and smaller problem spaces, i.e., smaller p_{max}, should have a positive impact on selector performance.

Selector generation is carried out by four different mechanisms: WEKA's components to generate a `J48` decision tree or an `M5P` model tree (sec. 6.2.1, p. 121), the winner-takes all (WTA) selector generator, and the `EnsembleSelectorGenerator` (sec. 6.2.3, p. 128). The `EnsembleSelectorGenerator` creates individual predictors for each algorithm in $\mathbb{A}$ by using a (configurable) auxiliary selector generator. The generated predictors are then stored together, and used as an *ensemble*. `M5P` is used as auxiliary selector generator in this case, as the model trees employ *linear* regression — they should hence fit optimally to the synthetic test setup described in section 8.2.1.

A random algorithm selector is used as an additional benchmark. Only selection mappings that outperform it are average-effective (see def. 2.1.5, p. 17). Random selection should perform as well as any winner-takes-all approach in this scenario, because all algorithms dominate equally large regions of the simulation space. Therefore, the maximal constant gain (def. 2.1.9, p. 19) is 1, i.e., on average no constant selection mapping works better than random selection. Furthermore, this implies that all average-effective selection mappings are also adaptive-effective (see def. 2.1.6, p. 18). Finally, an optimal selector that implements equation 8.1 is added, to check the effectiveness of the generated selectors. A similar evaluation approach, i.e., to compare generated selection mappings with constant ones (winner-takes-all) and optimal ones, has been used for other ASP solution approaches as well (e.g., in [338]).

8.2.2. Results

Figures 8.8 and 8.9 show the performance of the generated selectors for the validation set, i.e., the set where most models exhibit feature values that were not seen before. Sample models are drawn randomly as before, but now each feature value is multiplied by 10. The experiment duration is

[5] As the benchmark models are very simple and generate exactly the same load for the given parameters, three replications should suffice to get a good estimate of the average execution time.

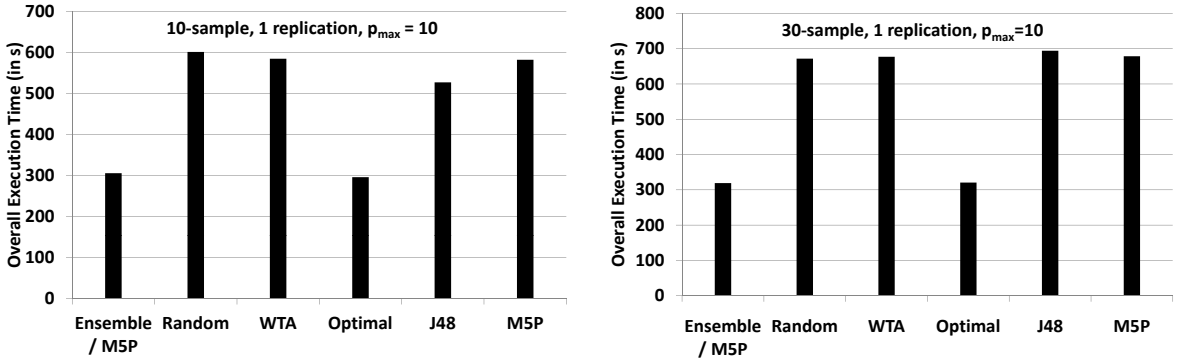

Figure 8.8.: Impact of exploration sample size on selector performance. Sample size does not play an important role in this synthetic performance, as a linear relationship could even be derived from two samples only. While the ensemble selector (`Ensemble / M5P`) achieves near-optimal performance, neither decision trees nor model trees are able to clearly outperform random selection.

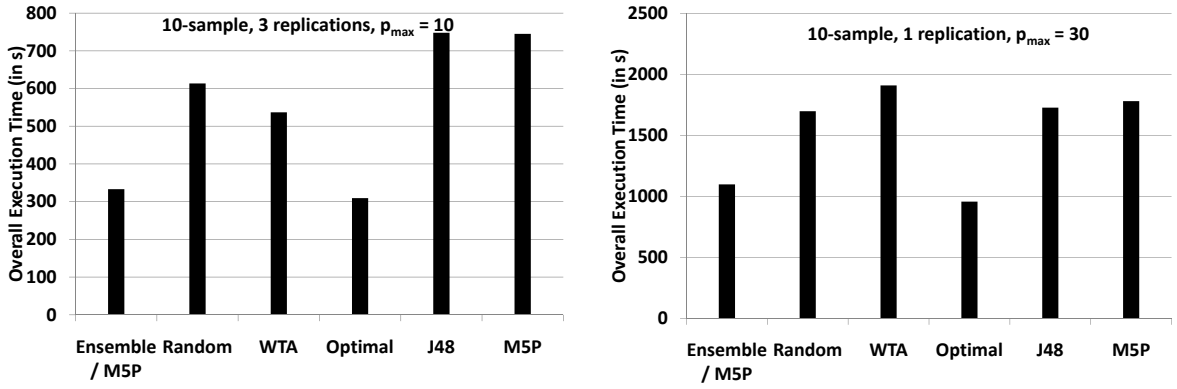

Figure 8.9.: Impact of replication number and problem space size on selector performance. While the number of replications has little impact on selector performance, increasing the size of the problem space also increases the gap between the best selector (`Ensemble / M5P`) and the optimum.

summed over the execution times of 20 models that have been drawn in this manner. All selectors are executed on the *same* randomly selected models.

The basic setup is defined with sample size $s = 10$, number of replications $r_a = 1$, and maximal model parameter value $p_{max} = 10$. It is shown on the left of figure 8.8. Each of the other plots shows corresponding results in case one of the parameters is tripled, i.e., $s = 30$, $r_a = 3$, or $p_{max} = 30$. In all scenarios, the ensemble selector consistently outperforms the other selectors and achieves near-optimal performance. Neither increasing the number of explored model setups (sample size s) nor the number of replications (r_a) has much impact on selector performance. This means that no selector benefits from additional efforts to explore the simulation space, at least not in terms of a larger sample (s) or in terms of less noise (r_a). Increasing the size of the problem space ($p_{max} = 30$, right side of fig. 8.9), however, leads to a slightly increased gap between ensemble selector and optimum. Even in this case, the selection efficiency (def. 2.1.7, p. 19) is still ≈ 0.87, i.e., the distance to the optimal throughput (achieved by S^*) is less than 13% on average. The maximal adaptation gain (def. 2.1.8, p. 19) in this case, i.e., the performance benefit of using an optimal selector instead of the best constant one, is ≈ 0.43. This means optimally adapting to the problem features results in 43% more throughput, i.e., an execution that is 43% faster on the average. While this speed-up is considerable, the case studies (ch. 9, p. 177, and ch. 10, p. 197) show that potential benefits can be much larger for real-world problems. However, neither J48 nor M5P are able to consistently outperform a random selection in

this scenario; they fail to deliver any performance benefit. This illustrates that there is **no guarantee of success** — and how important it is to test *multiple* selector generators.

Significance of Results

Note that the presented results merely show that a *suitable* algorithm selector (here, Ensemble/M5P, see fig. 8.9) that considers *suitable* problem features is able to achieve near-optimal performance. This means that the SASF, as a software framework for simulation algorithm selection, *can* be used effectively. In particular, the assumption that *all* performance-relevant model features have been identified is *very* optimistic — feature selection as such is a problem that should be addressed explicitly in the future (see def. 2.1.3, p. 15). The quality of the generated selection mappings and the efforts required to construct them also depend strongly on the available SASF components: realistic application domains may require new algorithms for simulation space exploration, adaptive replication, selector generation, or selector evaluation. Now that the SASF has been shown to function effectively, the next section revisits the other requirements discussed in section 4.1 (p. 79) and section 4.3 (p. 89).

8.3. Revisiting the SASF Requirements

8.3.1. Use Cases & User Interfaces

The discussion on SASF use cases (sec. 4.1, p. 79) identified several roles of potential SASF users: experimenter, performance analyst, developer, and deployer. The main goal of the SASF is to support experimenters in conducting simulation experiments; the other roles are either related to this task (performance analyst, developer) or required to support it (deployer). Since experimenters are not necessarily experts in the field of simulation, the user interface for experimenters should be kept as simple as possible. This requirement has been partly accomplished by integrating algorithm selection into the `AlgoSelectionRegistry` in a transparent manner — the user interface does not change at all. If an experimenter leaves the simulator configuration unspecified, this can now be filled in automatically. On the other hand, no specific graphical interface to support algorithm selection has been developed yet; it is necessary to let the user decide among the performance metrics of interest. Other components an experimenter might be confronted with are the adaptive simulation runner and its sub-system for simulation algorithm portfolio selection (sec. 7.2, p. 134). While the former is as easy to use as any other simulation runner available in JAMES II, the latter merely requires some standard parameters to be called (e.g., the URL of the performance database).

Interfaces for developers, performance analysts, and deployers are all on a programming level, i.e., they are the Java interfaces provided by the SASF sub-systems (e.g., the SPDM). Developers are supported by the algorithmic change evaluator (sec. 7.3.4, p. 156), which is a simple Java class. Performance analysts are supported by the simulation space explorer (sec. 7.3.3, p. 153), which can be combined with the simulation end time calibrator and the adaptive simulation runner. This flexibility makes the simulation space explorer rather complex to configure correctly. Deployers, finally, may use the SPDM to generate and evaluate selectors. They simply call the `SelectorGeneratorEvaluation` (see fig. 6.12, p. 128). The selector they deem most suitable can be deployed to the `AlgoSelectionRegistry` with another simple call (and the instantiation of a corresponding `SelectorManager`).

8.3.2. Technical Requirements

Based on figure 4.6 (p. 97), figure 8.10 (p. 172) shows which of the SASF components realize which required functionality. The figure also shows how different SASF concerns — performance data storage, data analysis, and so on — have been separated from each other.

Table 8.1 summarizes the requirements listed in table 4.1 (p. 92) and names the SASF components that address them the most. The SASF scales with the number of algorithms by managing their

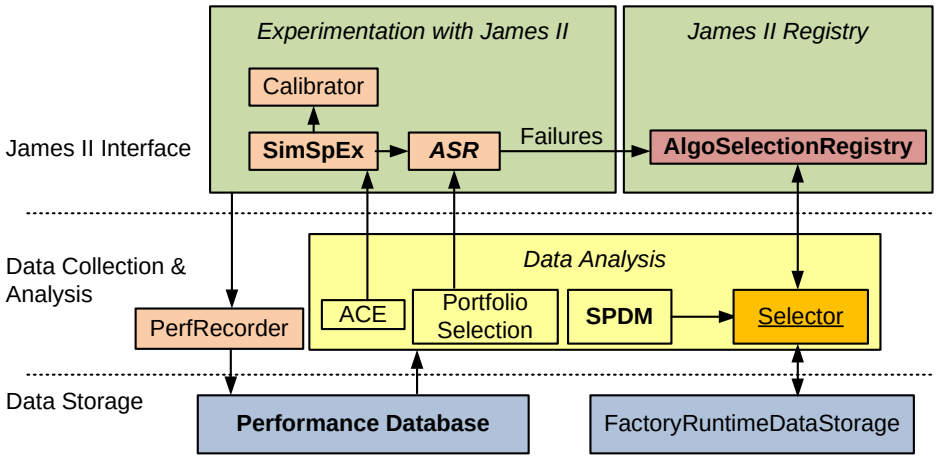

Figure 8.10.: Overview of the developed SASF components and their interrelationships, based on the general SASF design shown in figure 4.6 (p. 97). Color semantics are the same: data storage is blue, data analysis is yellow, performance exploration is orange, and knowledge exploitation is red. Major components have bold names; explicit ASP solutions are produced by the SASF in form of algorithm selectors (underlined, dark yellow). The implementation shown here partly differs from the planned design: the failure reporting of the adaptive simulation runner has not been considered in chapter 4 (p. 79). Furthermore, the adaptive simulation runner (ASR) *combines* performance exploration and knowledge exploitation.

performance data in a performance database. Its reference implementation uses the Hibernate persistence system. Since this can be used with various database management systems, the performance database should scale well even for many algorithms (and their performance data). Another important mechanism in that regard is the sub-system for simulation algorithm portfolio selection: it allows to focus on relevant selection trees, i.e., the runtime configurations of JAMES II that really matter (see sec. 5.1.1, p. 104). The `AlgoSelectionRegistry`, finally, scales with the number of algorithms because it selects complete runtime configurations at once, and therefore avoids a regeneration of all selectors in case a new algorithm is added (see discussion in sec. 4.2.3, p. 87).

Behavioral introspection is automated by using the sub-system for simulation space exploration (including adaptive simulation runner and simulation end time calibration), and recording the observed performance to the performance database. The algorithmic change evaluator illustrates the potential of automatic introspection by supporting developers in the assessment of code changes. Scalability with respect to performance data is not only provided by the performance database, but also by the SPDM. It integrates dedicated data mining tools, i.e., tools that should be able to cope with large amounts of performance data. Behavioral intercession is supported by two independent mechanisms. The SPDM allows to analyze performance data by generating a suitable selector for the `AlgoSelectionRegistry`, which adapts the registry's selection behavior. Alternatively, the adaptive simulation runner changes the behavior of the simulation system during replication.

The performance evaluation of selectors is only partially supported so far. While the SPDM includes several evaluation strategies and selector performance metrics (see sec. 6.2.2, p. 123), meta-learning has not been included yet. A component to support this in future would be the `SelectorEnsemble`, as discussed in section 8.1.3 (p. 164).

8.3.3. Summary

The results presented in section 8.2.2 show that — given a set of suitable SASF components (e.g., feature extractors and selector generators) as well as a set of simulation algorithms where none dominates the others — it is possible to successfully use the methods introduced in the previous chapters.

Requirement	**SASF**	Components
Separation of Concerns	✔	-
Scalability w.r.t. #Algorithms	✔	Performance Database, Portfolios, `AlgoSelectionRegistry`
(Automated) Behavioral Introspection	✔	Simulation Space Exploration (Simulation End Time Calibration, Adaptive Simulation Runner, Algorithmic Change Evaluator), `PerfRecorder`, Performance Database
Scalability w.r.t. #Performance Data	✔	Performance Database, SPDM
Behavioral Intercession	✔	SPDM, Adaptive Simulation Runner, `AlgoSelectionRegistry`
Meta-Learning / Performance Evaluation	≈	SPDM, `AlgoSelectionRegistry`

Table 8.1.: Revisting the requirements summarized in table 4.1 (p. 92). Checkmarks denote the features that the SASF complies with. Only meta-learning is currently not supported by a dedicated mechanism, while the performance evaluation of selectors is done by the SPDM. However, the foundation for meta-learning has already been laid in the `AlgoSelectionRegistry`.

It is now possible to:

1. Conduct meaningful automated performance experiments.
2. Record and store observed performance data.
3. Generate adaptive-effective selection mappings.
4. Deploy the selection mappings to a prototypical registry that supports automatic simulation algorithm selection.

Alternatively, section 7.2 (p. 134) shows how to employ automatic selection mechanisms even in the *absence* of this overall process: if no information is available, stochastic simulation can still be sped up by adaptive replication. If no meaningful model features can be extracted but performance data is available, this approach can be enhanced by portfolio selection. While all these findings show the potential of the presented methods, they do not show their impact in more realistic scenarios. This is what the third part of this thesis focuses on.

Part III.

Examples and Conclusion

9. Case Study I: Chemical Reaction Networks

> *For an exhaustive molecular reconstruction of the activity of a single cell, we would need to simulate the interactions of around* 10^{12} *such molecules. And we would have to continue the simulation many seconds, minutes, hours, days, or even years, a span of time scales of around* 10^{15}*. This would require unimaginably large computational resources — around* 10^{27} Blue Genes*. There simply won't be enough stuff in the whole solar system to build such monsters. Yet, this would be only the beginning of our problems.*
>
> Denis Noble [242, p. 76]

This chapter shows how the methods developed in chapter 4 to 8 can be applied to stochastic simulation algorithms (SSA), which have already been discussed briefly in section 1.3.1 (p. 4). JAMES II offers several kinds of simulators for the field of computational systems biology (see [308]). Some of them — e.g., those for models expressed in stochastic π-calculus [200] — rely on SSA implementations as well. At the same time, the performance of different SSA implementations is still fairly unexplored, particularly when it comes to different model properties and different sub-algorithms, e.g., RNGs or event queues (see discussion in [158]). Their relative merits are even debated in the literature (e.g., [281, p. 21]). Therefore, applying the algorithm selection methodology developed in part two of the thesis seems to be particularly beneficial in this setting.

The following case study is focused on *exact* SSA variants, i.e., methods that strictly adhere to the so-called *chemical master equation (CME)*. The CME is a differential equation defined on a probability function; it defines the behavior of a chemical system in case it is well-stirred and in thermal equilibrium [107]. Stochastic simulation is used to approximate the CME, as it is often intractable by current analysis methods. Exact SSAs compute trajectories that comply with the CME, while approximative SSA variants, such as τ-leaping (see sec. 1.3.1, p. 4), do not guarantee this.

Clearly, the accuracy of simulation algorithms is a common performance metric and can be considered in the selection process (see sec. 5.1.1, p. 108). Doing so, however, is particularly challenging in case of SSA. This is because measuring the accuracy requires two *sets* of trajectories: one generated by an exact SSA variant, and one from the approximative variant under scrutiny. A statistical test can now be used to reject the null hypothesis that claims both trajectories have been sampled from the same population. The problem is detailed in a previous study that investigates the performance of many SSA configurations available in JAMES II, and which also used the performance database presented in chapter 5 (p. 101) [158, p. 222]. Requiring two sample sets makes measuring the accuracy of SSAs quite expensive: numerous simulation runs for all considered variants are necessary before the statistical test can be applied — and the result will merely give the accuracy for a *single* simulation problem. Although approximative methods are (usually) much faster than their exact counterparts, the uncertainty regarding their accuracy for a given model is a major drawback. The previous study on JAMES II SSAs revealed that such accuracy loss may even occur for rather simple benchmark models [158]. Finding out how model properties affect the performance of approximative SSA variants in terms of accuracy is a very interesting topic; nevertheless, it is beyond the scope of this chapter.

Another limitation of this study is the restriction to a single benchmark model — the cyclic chain system (CCS) described in section 7.3.1 (p. 149) — where reactions always occur at comparable frequencies, i.e., the modeled systems are not stiff. *Stiff systems* are formally defined as *"[s]ystems with eigenvalues whose real parts are widespread along the negative real axis [...]"* [34, p. 39]. While the

previous quote relates to models of continuous systems, similar scaling issues occur when these are translated to discrete-event models. Put simply, stiff systems are hard to simulate because their dynamics occur on very different time scales: some variables of the system's state change only rarely, while others change quite often. This makes efficiently simulating stiff systems a challenging problem, since the more frequent changes may slow down the simulation considerably. This motivates specific simulation methods that deal with multiple time scales, e.g., [27, 299]. An investigation of stiff reaction networks would have to involve these methods as well, but they are not available in JAMES II yet.

Both this and the next chapter are structured similarly: the first section will briefly survey the main implementation differences among the simulation algorithms under scrutiny, and will also highlight the usage of auxiliary algorithms, i.e., plug-ins. The second section will then evaluate some of the mechanisms that have been developed in the previous part. As the SSAs are among the most stable, well-engineered, and relevant simulators in JAMES II, this chapter will cover some more methods than chapter 10. Chapter 10 mainly serves to demonstrate the generality of the presented methods.

9.1. Algorithms under Consideration

A chemical reaction network as processed by the SSAs basically consists[1] of a set of reactions, e.g., of the form

$$R_1 : A + B \xrightarrow{r_1} C \tag{9.1}$$

where r_1 is the rate constant that basically determines how likely particles of species A and B react to C when they encounter each other. Given the amount of the reactants (i.e., A and B), r_1, and the volume of the system, one can now calculate the (exponentially distributed) rate with which this reaction occurs *somewhere* in the system. The rate parameter is also called the *propensity* of the reaction. As mentioned in section 1.3.1 (p. 4), this now allows to formulate the overall problem as a continuous time Markov chain: given the current state of the model, i.e., the amount of particles for each species, the reaction propensities of all reactions denote the transition probabilities of the Markov chain to potential future states. Since reactions (usually) *change* the state of the model, and the reaction propensities rely on the state, these have to be recalculated after a reaction occurred. The simulation now proceeds in a discrete-event manner, by applying the effects of subsequent reactions to the current state until the simulation end time is reached.

Direct Method (DM)

The *direct method (DM)* is presented in the original paper by Gillespie [105], in which he introduces the mathematical model on which the SSAs operate. The direct method sums up all reaction propensities and uses this sum as the parameter of an exponential distribution. The drawn random number determines when the next event occurs in the model. Afterwards, the method has to decide *which* reaction occurred. This is done by randomly drawing one of the reactions, in proportion to their propensity. After the time point of the next reaction and the *kind* of the next reaction are known, the direct method simply executes the reaction, updates the state accordingly, and recalculates all reaction propensities. Then it starts over.

First Reaction Method (FR)

The *first reaction method (FR)* is also presented in [105]. Instead of determining at first the time of the next event and then the kind of reaction to occur, it starts out with calculating the time of next occurrence for *each* reaction individually. This is done by using the reaction propensity as the

[1] What follows is a *very* superficial explanation that shall merely highlight the algorithmic differences. The overall subject matter is much more complicated than that, e.g., when it comes to the calculation of stochastic reaction rates from deterministic ones.

parameter of an exponential distribution and drawing a random number. After occurrence times have been computed for each reaction, the reaction with the smallest occurrence time is selected for execution. Updating the state and the reaction propensities concludes the iteration.

Next Reaction Method (NRM)

The *next-reaction method (NRM)* by Gibson and Bruck [104] extends the first-reaction method in several ways. Most importantly (from an algorithm selection perspective), the NRM relies on an *event queue* to sort reactions by their next time of occurrence. Similar to FR, the NRM then executes the reaction with the smallest time stamp. The reaction is now retrieved from the event queue. Instead of recalculating the propensities of *all* reactions, as done by the FR method, the NRM now restricts the recalculation to those propensities that could have changed. Imagine a model consisting of the reaction defined in equation 9.1, as well as the reactions $R_2 : C \xrightarrow{r_2} D$ and $R_3 : D \xrightarrow{r_3} E + 2F$. If the first reaction (eq. 9.1) is executed, only the propensity of R_2 needs to be updated — as there is now one more particle of species C (the product of R_1). The propensity of R_3 remains the same, and is only changed if the number of D particles is changing. During initialization, the NRM creates a directed *dependency graph* that connects each reaction with those it influences. After a reaction R_i is executed, the propensity is only recalculated for reactions that are influenced by R_i, i.e., the reaction that was executed. The time of next occurrence can be scaled by the ratio of old and new propensity, so that only one new random number has to be drawn per iteration: it determines the next occurrence time for R_i. Afterwards, all involved reactions will be re-queued by the event queue.

JAMES II offers two alternative implementations of the NRM, in the following named NRM-A and NRM-B. In the previous study [158], NRM-B consistently outperformed NRM-A whenever each used its most suitable event queue.

Optimized Direct Method (ODM)

The *optimized direct method (ODM)* presented by Cao et al. in [31] enhances the direct method. Like the NRM, it uses a dependency graph to restrict propensity recalculation. This also allows to optimize the overall propensity summation of the direct method: only the recalculated propensities have to be accounted for, by subtracting the old ones and adding the new ones. Additionally, the reactions are sorted by their propensity (in descending order) after some pre-simulations have revealed their average occurrence frequencies.[2] This speeds up the DM when selecting *which* reaction occurs next, since the most likely reactions are now considered first.

Event Queues

The event queues that are considered here belong to the standard implementations provided by JAMES II [135]. For example, JAMES II provides a naïve solution, SimpleEQ, that is based on a sorted list. It is, surprisingly, a good choice for exact SSAs on some models [158, p. 224]. An adaptation of the popular MList [112] has also been included; its re-queuing operation has been improved, so it is called MListRe here. It previously performed well on some other SSA benchmark models. Additionally, a Heap-based event queue is used, as it has been discussed in [31, 104]. Other event queues without much relation to SSAs have been added for the experiments regarding the adaptive simulation runner, to make convergence more challenging. Most of them have already been discussed in [135]. Prior execution time observations show the strong dependence of the next reaction method on its event queue [158]. Since this dependency is strong enough to let the next reaction method be either quite slow or quite fast in relation to the other algorithms, this again shows the pitfalls of restricting performance analysis to only a few monolithic algorithms.

[2] In the JAMES II implementation, this is handled during a warm-up phase.

Random Number Generators

Cao et al. argue that the performance of the random number generator is negligible for SSA execution performance [31, p. 4064]. This finding is confirmed by the observations on SSA performance in JAMES II [158]. Therefore, the default Java implementation (a linear congruential generator, see eq. 3.1, p. 64) is used in the following experiments.

9.1.1. A Sample Approach to SSA Performance Analysis

In contrast to the parallel and distributed discrete-event simulators covered in chapter 10 (p. 197), the performance analysis of SSAs has not gained much attention so far. Besides empirical studies such as [158], a notable exception is the performance analysis of ODM presented by Cao et al. in [31]. However, neither their practical experience[3] nor their theoretical analysis enable simulation algorithm selection in practice. The analysis centers on the cost of various operations, where a chemical reaction network with M reactions is considered. Then, Cao et al. define a weighted average degree

$$D = \frac{\sum_{i=1}^{M}(d_i \cdot k_i)}{\sum_{i=1}^{M} k_i}$$

where d_i is the out-degree of reaction R_i, i.e., the number of reactions it influences (except itself), and k_i is the average number of executions per simulation (as determined by ODM via pre-simulation). They also define the variable $S*$ as the optimized search depth, i.e., it is the average number of summands until the reaction that occurred can be picked by ODM. The search depth is *optimized* here, as the reactions are ordered by previous occurrence frequencies, so that those which are likely to have a large propensity are in front of the list (as discussed before). The result of the analysis is the following rule: *"The only situation that ODM is less efficient than NRM is when $D \ll M$ and $S* \approx M/2$"* [31, p. 4065].

This illustrates why theoretical considerations — as useful as they are for investigating general bottlenecks — do not suffice to enable algorithm selection in practice: just *how* much smaller than M should D be? How to measure $S*$ before simulation, and when is it sufficiently close to $\frac{M}{2}$? All this depends on the available implementations, the hardware, and likely also on some other model properties (e.g., M).

9.2. Experimental Evaluation

This section presents empirical observations regarding the effectiveness of the developed methods, which are applied to the set of algorithms outlined in the previous section. After defining the experimental setup (sec. 9.2.1), three major scenarios of applying the SASF are considered:

- *Simulation algorithm performance evaluation* (sec. 9.2.2, p. 181): the methods presented in section 7.3 (p. 145) are applied to explore the performance of the stochastic simulation algorithms (see sec. 9.1) for the CCS benchmark model (see sec. 7.3.1, p. 149). The results are managed by the performance database presented in chapter 5 (p. 101). The data is also used to analyze the impact of code changes with the algorithmic change evaluator (see sec. 7.3.4, p. 156).
- *Algorithm selection without problem features* (sec. 9.2.3, p. 186): the adaptive simulation runner from section 7.2 (p. 134) is evaluated. The performance data obtained in section 9.2.2 is used to construct an algorithm portfolio for faster convergence (as discussed in sec. 7.2.2, p. 140).
- *Algorithm selection with problem features* (sec. 9.2.4, p. 191): the SPDM framework presented in chapter 6 (p. 115) is applied to the data collected in the first step (sec. 9.2.2). Afterwards, the generated selectors are evaluated with the methods presented in section 6.2.2 (p. 123), and

[3] *"For the practical problems we have tried, NRM is usually less efficient than DM."* [31, p. 4062]

then deployed to the `AlgoSelectionRegistry` (ch. 8, p. 159) to test them against some formerly unseen CCS configurations in practice.

It should be clear that this study is focusing on the merits and weaknesses of the SASF *as such*. Otherwise, more data from other carefully selected benchmark models — and not just a single one, the CCS — would have to be collected and analyzed. In other words, the following experiments shall illustrate the potential benefits of applying the SASF, and that it works as intended.

9.2.1. Setup

The following experiments have all been executed on the same workstation, in order to make results as comparable as possible. The workstation has 8 GB RAM and two quad-core Xeon CPUs (E5420) with a clock rate of 2.5 GHz. Overall, there are 8 CPU cores to execute jobs in parallel; no hyperthreading is available. Its Java SciMark [263] composite score is 803.46. The workstation runs the 64-bit edition of Windows XP Professional; JAMES II is executed by a beta-version of Sun's 64-bit Java 1.7 Runtime Environment (1.7.0-ea-b66). The cyclic chain system discussed in section 7.3.1 (p. 149) is used as a benchmark model, since it can be parameterized in many ways to mimic different kinds of chemical reaction networks.

9.2.2. Simulation Space Exploration

This section does not focus on algorithm selection as such, but on the experimentation mechanisms described in section 7.3 (p. 145), which are necessary to obtain the performance data required for both portfolio selection (see sec. 9.2.3, p. 188) and data mining (see sec. 9.2.4, p. 191).

Setup

In this scenario, the goal of the exploration mechanism is to explore the execution time of six SSA algorithms: DM, ODM, FR, and NRM (variant B), the latter being combined with SimpleEQ, MListRe, and a Heap-based event queue. The algorithms are evaluated on 150 parameterizations of the CCS benchmark model (sec. 7.3.1, p. 149). A full factorial experiment is defined on the CCS benchmark model as given in equation 7.8 (p. 149), i.e., all combinations of the following parameters are considered:

- Number of species: $N \in \{5, 10, 15, 20, 25\}$.
- Number of reactants/products: $k \in \{1, 2, 3, 4, 5\}$.
- Initial population: $X_i \in \{2000, 5000, 8000\}$.
- Factor for number of reactions: $r \in \{1, 2\}$.

The factor for the number of reactions, r, is not included in equation 7.8 (p. 149). If $r = 1$, all reactions are generated as defined by equation 7.8 (p. 149), but for $r = 2$ each reaction is defined *twice*, and so on. Increasing r therefore increases the overall number of events that occur in the system, but since it affects *all* reactions equally it does not affect the quasi-steady state property of the CCS. Figure 9.1 (p. 182) shows a sample CCS reaction network for $N = 4$, $k = 1$, and $r = 1$. The reaction rate constant c is set to $10^{-4 \cdot (k-1)}$, i.e., it is determined by the number of reactants involved per reaction (k). This levels out the reaction frequency in the model over different k, since it is calculated by multiplying the stochastic rate with the current amount of particles for each reactant (e.g., [106]).

Considering all combinations of the above parameter values results in $5 \cdot 5 \cdot 3 \cdot 2 = 150$ simulation problems.

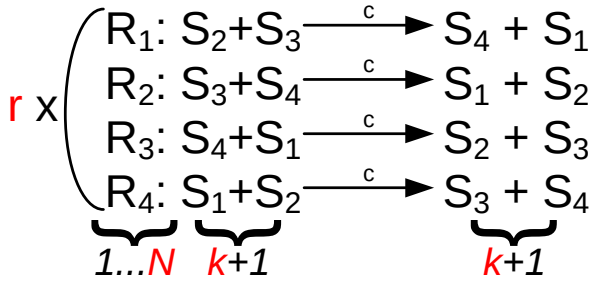

Figure 9.1.: A sample CCS model with $N = 4$, $k = 1$, and $r = 1$. All parameters that determine the structure of the reaction network — N, k, and r — are shown in red.

The impact of both the adaptive simulation runner (sec. 7.2, p. 134) and the calibration mechanism (sec. 7.3.2, p. 150) on performance exploration is evaluated by considering four different scenarios:

1. **Fixed end time (small), fixed replication:** The simulation end time is set to 10 s for all simulation problems. A fixed number of four replications is executed for each algorithm.

2. **Fixed end time (small), adaptive replication:** The simulation end time is again set to 10 s. Instead of executing four replications per algorithm, the same number of runs is conducted by a specific variant of the ϵ_n-DECREASING policy, which tries every algorithm at least once (in the beginning).

3. **Simulation end time calibration, adaptive replication:** The simulation end time is calibrated by the algorithm described in section 7.3.2 (p. 150), which is configured with $t_{opt}^{wct} = 5$ s, $t_{max}^{wct} = 10$ s, $t_{min}^{sim} = 0.1$ s, and $t_{max}^{sim} = 1000$ s. The subset of algorithms used for calibration consists of two elements: ODM and NRM (combined with MLISTRE). As in the second scenario, the variant of ϵ_n-DECREASING steers the adaptive replication.

4. **Fixed end time (large), fixed replication:** The simulation end time is set to 100 s for all simulation problems. A fixed number of four replications is executed for each algorithm.

All the above scenarios are executed sequentially, i.e., on a single core. This minimizes the additional noise that is introduced by multi-threading overhead.

Performance Results

Figure 9.2 (p. 183) compares the overall execution times of the four performance experiments. Clearly, the second scenario (adaptive replication with fixed simulation end time) is the fastest option (further details on adaptive replication performance are presented in sec. 9.2.3, p. 186). Both scenario one and scenario three achieve a similar speed, with slowdowns of $\approx 26\%$ and $\approx 43\%$ respectively. Scenario four, however, takes much longer to execute: it is more than eight times slower than scenario three, with ≈ 41 h instead of ≈ 5 h execution time.

One could now argue that the performance data obtained by scenario four is more accurate, since more time has been spent to execute the algorithms under investigation — but this is not the case. The average simulation end time used in scenario three (which uses calibration) is 222.27. In other words, the calibration mechanism leads to simulating the models for much more than 100 s simulation time *on average*. Yet, the *execution time* of scenario three is much *smaller* than the execution time of scenario four, because it treats each simulation problem *individually* (by calibrating the simulation end time). Moreover, figure 9.2 shows that the overhead introduced by calibration is negligible when compared with the overall run time of the performance experiments.

To further illustrate the effect of simulation end time calibration and adaptive replication, figure 9.3 (p. 184) displays the execution times of individual simulation runs for each scenario, sorted in ascending

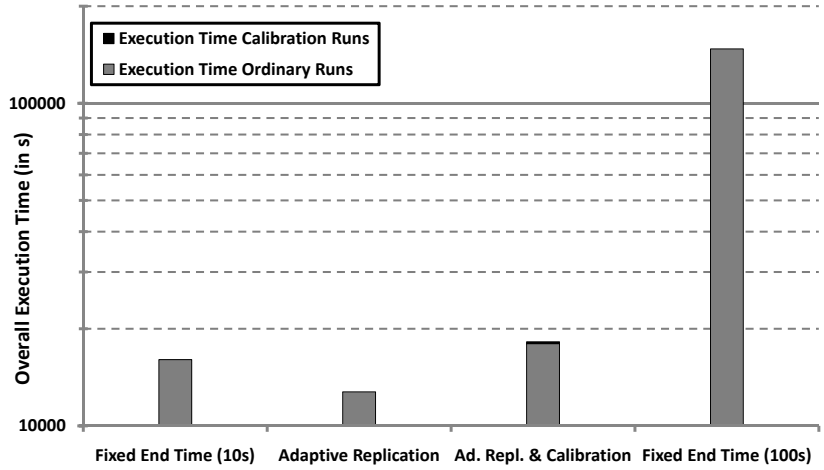

Figure 9.2.: Overall execution times of the four performance exploration scenarios. Only the third scenario (adaptive replication & calibration) includes a non-zero overhead due to simulation end time calibration (black).

order. The execution times in the third scenario (lower left plot) are all relatively close to each other, the average execution time is ≈ 4.28 s. It is relatively close to the desired value t_{opt}^{wct}, which was set to 5 s (see sec. 9.2.2, p. 181). The difference stems from some execution times that are considerably smaller than 5 s. For some simulation problems (20 out of 150), the maximal simulation end time ($t_{max}^{sim} = 1000$ s) is still too small to keep the algorithms busy for about 5 seconds (wall-clock time). On the other hand, some algorithms exceed the maximum wall-clock time ($t_{max}^{wct} = 10$ s) and are automatically censored by the calibration mechanism, i.e., their execution is aborted after 10 s of wall-clock time. This further decreases the execution time of scenario three, at the cost of not always knowing the worst-case performance. There is no simulation problem for which all algorithms are censored.

Finally, note that figure 9.3 (p. 184) shows $150 \cdot 6 \cdot 4 = 3600$ execution times (simulation problems $\times$ algorithms $\times$ replications) for scenario one, two, and four — but for scenario three (lower left) there are even 3900 execution times. This is because of the calibration mechanism: before a simulation end time is found and the replication begins, both algorithms from the test subset (here: ODM and NRM with MListRe) have already been executed by the calibrator with the given end time. This amounts to $2 \cdot 150 = 300$ surplus samples of execution time.

While the duration of scenario four (see fig. 9.2) is clearly too long and more suitable execution times can be obtained by simulation end time calibration, it is not yet clear if simulation end times larger than 10 s are indeed necessary. Maybe a shorter simulation time interval of 10 s (as used in scenario one and two) is sufficient to get an accurate picture of algorithm performance? Figure 9.4 (p. 185) shows that this is not the case. The outcomes from scenario three and one differ significantly. With the calibrated simulation end times, only two algorithms — ODM and DM — are able to outperform the competition (dark blue bars). For a fixed simulation end time of 10 s, however, *all* algorithms are able to outperform the others for at least one problem. This is due to extremely small execution times (see upper left plot in fig. 9.3, p. 184), so that stochastic noise governs the outcome. Similar differences occur for the worst-performing counts. These results illustrate the importance of methodologically sound performance experiments — if the data obtained from scenario one would be used to construct selection mappings, the outcomes are likely to be inconclusive.

Finally, note that the presented results are also interesting in itself. In a previous study [158], which did not consider the optimized direct method (ODM), several NRM variants could prevail instead. The good performance of ODM on the given CCS setups is particularly impressive as the ODM is explicitly optimized towards systems where a large difference in the occurrence probabilities of the

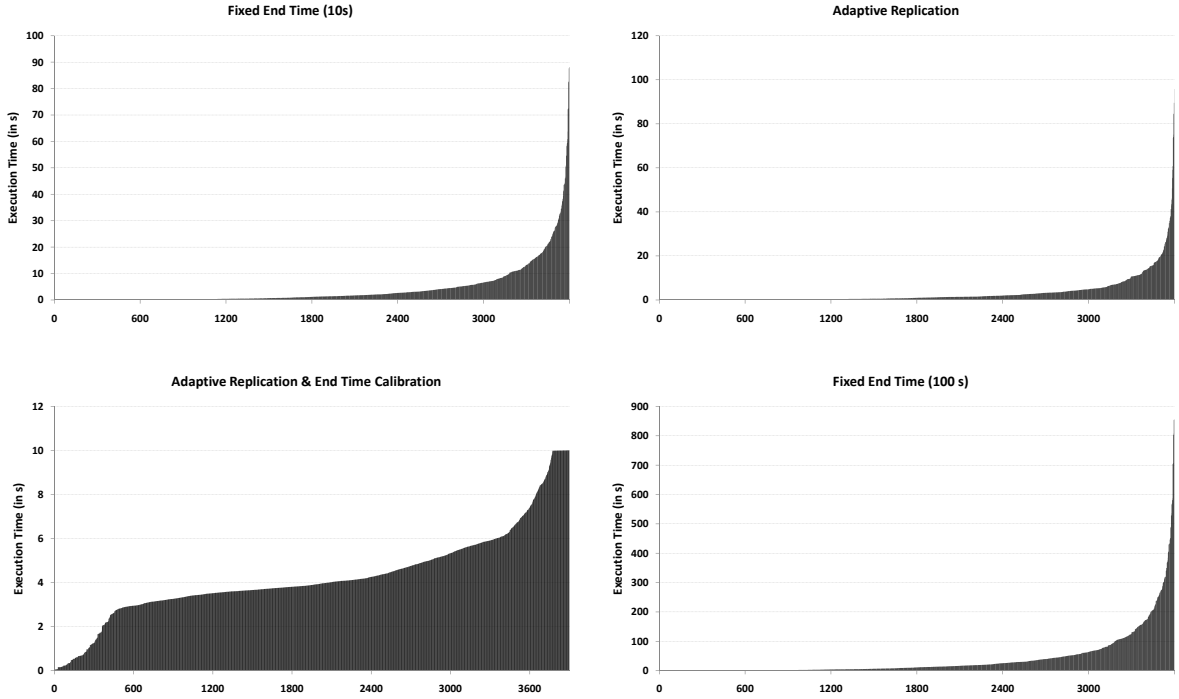

Figure 9.3.: Execution time distributions for the four performance exploration scenarios (upper row: one and two, lower row: three and four): the plots of the fixed simulation end time scenarios (i.e., all except the lower left plot) show that the execution times of individual simulation runs span several orders of magnitude. The upper right plot shows the impact of adaptive replication on the exploration (scenario two), as there are less longer runs than in scenario one. Note the different scales of the plots for scenario three (lower left plot) and scenario four (lower right plot).

reactions exist. This assumption does not hold for CCS. Furthermore, the data shows that a Heap may not be the best data structure to implement the NRM (as is suggested in [104]) — a finding that was already discussed in [158].

Development Support

Collecting and analyzing performance data in order to make informed decisions on algorithm development has already been motivated in section 4.1 (p. 79). The algorithmic change evaluator (ACE) presented in section 7.3.4 (p. 156) illustrates how performance data can be used for iterative software development. Its application is exemplified by relying on the performance data collected in scenario three (i.e., with simulation end time calibration). The code change to be evaluated affects the ODM simulator, which so far performed best overall (see fig. 9.4) but is now initialized with a complete dependency graph. This means the algorithm now updates reaction propensities even though these do not change — it should be slower, and the ACE should help to quickly detect this change for the worse.

The ACE is configured to use the same variant of the ϵ_n-Decreasing policy (sec. 7.2.1, p. 137) as discussed before, i.e., at first each algorithm is tried once. The set of algorithms to choose from is very small, as there is only one selection tree (i.e., runtime configuration) that contains the ODM; it is not necessary to re-evaluate its performance when combined with different algorithms. For comparison, the best runtime configuration that does *not* use the ODM is included for every simulation problem. The ϵ_n-Decreasing policy hence only has to decide between two alternatives. All this is set up automatically by the ACE, which also identifies all relevant problems from the performance database, i.e., those for which past performance data on the ODM is available. It creates new versions

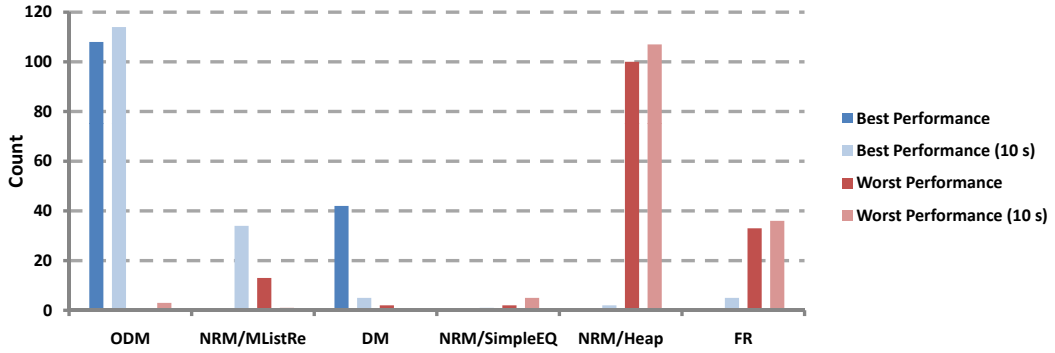

Figure 9.4.: Best/worst performance counts for SSA algorithms. Blue bars denote the count of problems where the given algorithm outperformed all others, red bars denote the count of problems where the given algorithm performed worst. Counts from scenario three (using calibration) are dark blue and red, counts from scenario one (fixed simulation end time of 10 s) are light blue and red.

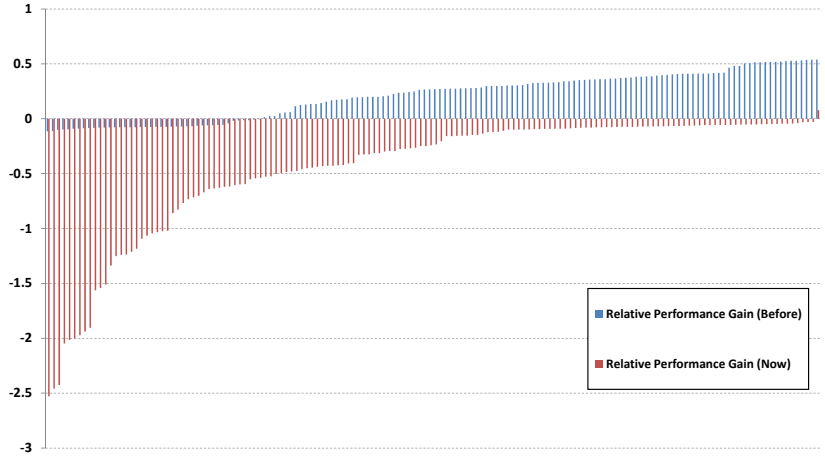

Figure 9.5.: Relative performance gains before and after the code change. Note that, for the sake of clarity, the performance gains are sorted independent of each other in ascending order, so two bars at the same position do *not* represent the same problem.

of all runtime configurations that include the changed algorithm, and marks all of its old runtime configurations as out-of-date within the performance data base (see sec. 5.1.1, p. 106). By doing so, one can easily build a 'performance history' for a certain algorithm, e.g., to assess optimization efforts.

The ACE is set up to execute six replications per problem. This results in $6 \cdot 150 = 900$ simulation runs (replications $\times$ simulation problems), which take ≈ 55 minutes. It is hence much faster than repeating the whole performance experiment from scratch (which takes more than five hours at best, see fig. 9.2, p. 183). The output of the ACE is used to calculate the relative performance gain regarding execution time, for all 150 simulation problems: it is the difference between ODM's average execution time (E_{ODM}) and the average execution time of the best alternative (non-ODM) setup (E_{alt}) for a given simulation problem, set into relation with the best overall execution time:

$$\frac{E_{alt} - E_{ODM}}{\min(E_{alt}, E_{ODM})} \tag{9.2}$$

A positive relative performance gain denotes setups where ODM delivers the best overall performance, as it is faster than the best alternative. The opposite holds for negative performance gains. Considering relative gains avoids a bias towards problems with longer execution times (and hence

larger absolute run time differences). It cannot be guaranteed in general that the performance of the other algorithms has not been changed in the meantime, e.g., due to more general changes in JAMES II that affect the overall runtime, or a slightly different experimental setup. Hence, the most trustworthy results are gained by re-evaluating *both* the new algorithm and the best alternative, and to only compare execution times recorded at the same time. By doing so, the only assumption that needs to be true relates to the performance *ranking* of the algorithms: the alternative setup used for comparison in equation 9.2 should still be the *best* alternative setup.

Figure 9.5 (p. 185) shows the relative performance gains (eq. 9.2) for the old and the new ODM variant. Clearly, the new variant performs much worse. While the old ODM variant is the best-performing algorithm for more than a hundred problems (see fig. 9.4, p. 185), i.e., its performance gain is positive on these instances, the new one has lost the lead on most of them. The code change should to be dismissed.

Besides this simple test run of the ACE, there is also anecdotal evidence that illustrates the benefits of maintaining performance data and re-evaluating algorithms against it. For example, a 'real' bug in the ODM implementation was only identified after reviewing previous performance data, which was recorded before the erroneous code change was made.

Summary

This section illustrated the potential of automating and improving the process of obtaining performance data for simulation algorithms, with a focus on execution time. Simulation end time calibration may reduce performance experiment duration significantly, as it avoids losing time by simulating benchmark problems for too long. Experiments using calibration will not only be faster, but also more precise, as the comparison in figure 9.4 (p. 185) illustrates. Consequently, the performance results gathered in scenario three (which employed simulation end time calibration) are reconsidered and analyzed in the next sections.

Adaptive replication helps to focus on the best-performing algorithms and yields additional speed-up; its performance is scrutinized in the next section. Finally, test runs with the algorithmic change evaluator show how previous performance data can be used to get quick feedback for programming decisions (fig. 9.5, p. 185).

9.2.3. Adaptive Replication

To test the adaptive simulation runner, the policies described in section 7.2.1 (p. 136) have been applied to execute a simple JAMES II experiment. The experiment comprises six CCS setups that are simulated for 25 seconds of simulation time. Each setup is configured to be replicated 500 times.[4] The fixed CCS parameters are:

- Initial population per species: $X_1 = \ldots = X_N = 10^5$.
- The number of reactants per reaction, k, is set to 3.

In this experiment, the overall number of species is varied between three and five, i.e., $N \in \{3, 4, 5\}$, and the factor for the number of reactions, r, is chosen from $\{1, 2\}$ (hence $3 \cdot 2 = 6$ CCS setups).

The adaptive simulation runner is restricted to four of the eight available cores, in order to minimize the impact of external load (e.g., from the operating system) on the results. The convergence of a policy can be easily studied by measuring its regret, i.e., the reward difference between an optimal choice and the policy's choice after each round (see sec. 2.3.2, p. 32). A per-round comparison is interesting because it shows how *fast* each policy is able to converge to the optimal solution: does it outperform the average case (of choosing randomly) already after 10 replications, or does it take more than 100 rounds? Policies that converge faster can be applied to a broader range of problems,

[4] This number seems fair, as there are real-world simulation studies using SSAs for many more replications (e.g., 3000 in [295]) and the number of algorithms to explore here is rather high (37, as discussed in the following).

i.e., also to those which require fewer replications. As discussed in section 7.2 (p. 134), the reward is usually a consumptive performance measure, i.e., a measure that refers to the consumption of a resource and should hence be minimized (see sec. 5.1.1, p. 108). The execution time of a simulation run is considered as reward in the following.

Instead of analyzing absolute regret, the central figure of merit is the *relative overhead* of not knowing the most suitable algorithm. The relative overhead o_n^p of using policy p to execute n replications is defined as

$$o_n^p = \frac{\sum_{i=1}^{n} reward_p^i}{n \cdot reward_{opt}} - 1 \tag{9.3}$$

where $reward_p^i$ is the reward received by policy p at round i, and $reward_{opt}$ is the expected reward of the optimal choice (which can be estimated). Subtracting 1 lets o_n^p denote the *overhead* only. Hence, an optimal solution — knowing the best algorithm from the beginning — would have an overhead of 0. Since execution time is used as reward here, $o_n^p = 0.1$ means that using policy p requires 10% more execution time than the optimal case for executing n replications. Note that equation 9.3 assumes policies to aim at *minimizing* the reward (see sec. 7.2, p. 134).

Rewards arrive asynchronously at the adaptive simulation runner (see sec. 7.2, p. 134), so the empirical measurements have first to be recorded and then analyzed post-mortem, i.e., after all runs have been executed. The collected data is also used to determine which of the options is indeed the best choice for each of the six CCS setups, so that an estimation of $reward_{opt}$ can be determined.

In this setup, the adaptive simulation runner has to choose from 37 options (see sec. 9.1, p. 178): a single setup for DM, FR, and ODM, in conjunction with the two NRM variants, each being combined with 17 different event queues ($1 + 1 + 1 + 2 \cdot 17 = 37$).

The policies described in section 7.2.1 (p. 136) have been mostly initialized with well-studied parameters. IntEstim and IntEstimDec are configured with $\alpha = 0.05$, a medium value also used in [316, p. 10]. SoftMax is initialized with a temperature $\tau = 0.1$ (see eq. 7.5, p. 139), also a medium value evaluated in [316, p. 10]. RewardComparison has been configured with $\alpha = \beta = 0.5$. The β parameter of Pursuit is set to 0.01 (as used in [298, p. 44]). ϵ_n-Decreasing (see sec. 7.2.1, p. 136) is initialized with $c = 0.2$ (a medium value investigated in [11]) and $d = 0.5$, the middle of d's permissible interval of $(0, 1)$. All other ϵ-policies have been initialized with $\epsilon = 0.15$, which has also been considered in [316, p. 10]. The α parameter of UCB2 has been set to 0.001, which was found suitable in [11].

Results Analysis

The relative overheads for the sixth setup ($N = 5, r = 2$) are shown in figure 9.6 (p. 188). This CCS setup imposes the most computational load. Figure 9.6 illustrates several important points: firstly, most policies are effective in this case, as their relative overhead is significantly less than the overhead of the random selection policy. This becomes apparent after ≈ 120 replications; for less replications the overhead is too noisy to be evaluated by a single experiment execution.[5] Secondly, policies from the UCB family as well as the SoftMax policy perform rather bad: UCB1 even performs *worse* than random selection. This illustrates how important it is to choose a *suitable* policy for the adaptive simulation runner, and that it is *not* sufficient to rely on asymptotic regret bounds — all UCB policies are proven to *eventually* converge to the optimum [11] (see sec. 7.2.1, p. 136). Clearly, a single experiment of this scale does not allow to draw any general conclusions on policy convergence — yet it shows that most policies are able to adapt quickly here, and that their convergence speeds may differ considerably.

The execution times of the policies over the whole experiment are depicted in figure 9.7 (p. 189). The worst case requires 3.89 times more time than the best case, even for these relatively small and simple CCS setups. The maximal attainable speed-up with respect to the average case (represented

[5] In the beginning, most policies have to 'guess' which option to choose and 'bad luck' leads to peaks in the relative overhead (see eq. 9.3, p. 187). The peaks vanish with a growing number of replications.

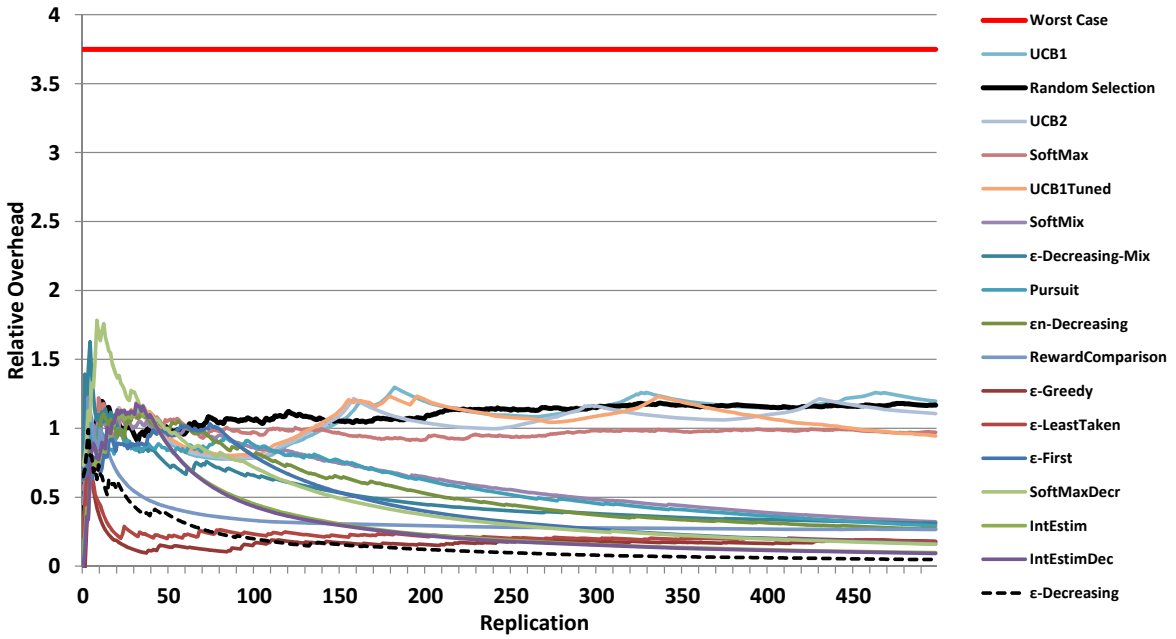

Figure 9.6.: Relative overheads of the adaptive simulation runner, relying on the given policies to execute one of the six CCS setups. The legend on the right also reflects the final order of the policy overhead after 500 replications (top to bottom), i.e., UCB1 performs worse than random selection but the other policies perform better. The worst case scenario has been calculated afterwards, by considering the average performance of the worst algorithm.

by RandomSelection) is ≈ 1.9, i.e., knowing the most suitable algorithm for each setup in advance would allow to almost halve the overall execution time. The best-performing policy — IntEstim — is only 7.7% slower than optimum, it yields a speed-up (w.r.t. RandomSelection) of 1.77: the overall execution time is reduced from 5.72 hours to 3.23 hours.

Furthermore, the performances of most policies are rather similar: all policies up to (and including) Pursuit yield execution times only 7.7% to 23% worse than optimum. The other group of similar-performing policies — consisting of SoftMax and the UCB policies — performs rather badly, they are not much better than the average case. Still, even the worst-performing policy (UCB2) needs 6% *less* time than RandomSelection. Moreover, note that using a *static* selection decided by a non-expert user only yields RandomSelection's performance *on average* — so one could argue that even using RandomSelection is preferable, as its performance *converges* to the average case: it is less risky than picking a single algorithm at the beginning.

The two clusters of similar-performing policies — one close to the optimum, one close to RandomSelection — represent the two possible outcomes of using the adaptive simulation runner: either its policy can identify a suitable algorithm, or it does not. In the latter case, this might be due to the policy, its parameters, or the shape of the algorithms' runtime distributions. This shows why both potential outcomes should be taken into account for portfolio selection, and are therefore reflected in the fitness function for the GA-based selection mechanism (eq. 7.6, p. 144).

Combination with Portfolio Selection

As the compilation and evaluation of a truly representative set of SSA simulation problems is beyond the scope of this thesis, the performance experiment discussed in the following will be restricted to showing the effectiveness of portfolio selection in a very limited and simple case, namely the adaptive replication of the CCS benchmark. A more thorough evaluation of portfolio selection effectiveness,

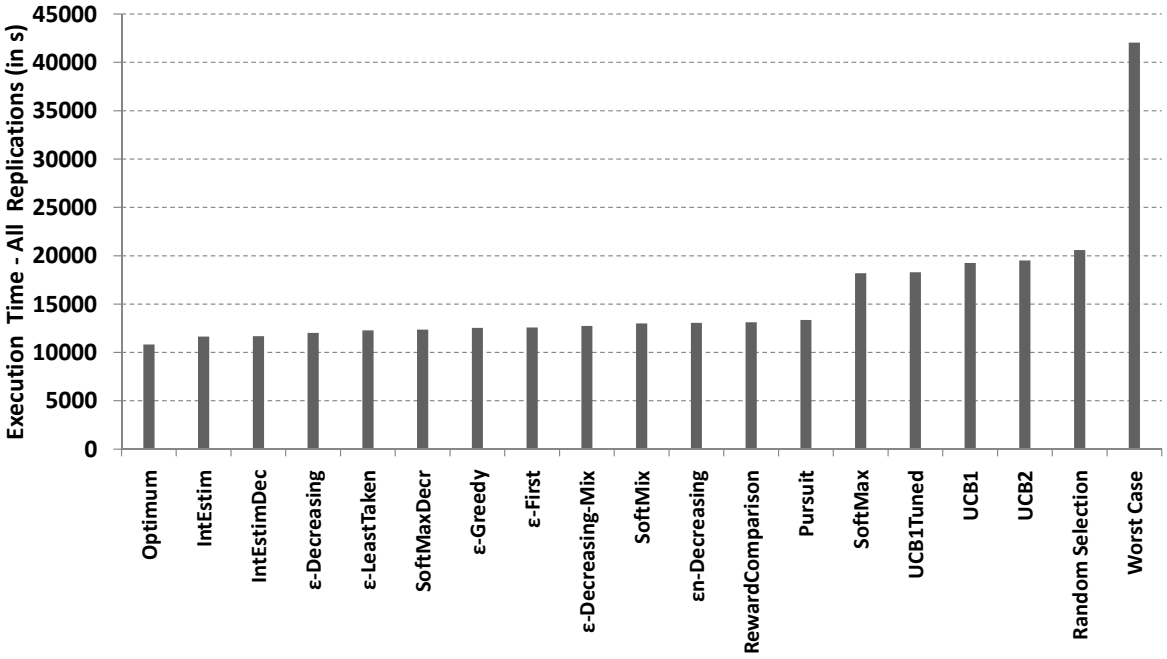

Figure 9.7.: Overall execution time of policies applied to execute the CCS benchmark experiment. Optimum and worst-case times have been estimated, just as in figure 9.6 (p. 188). Over the course of the entire experiment, even the worst-performing policy (UCB2) outperforms a random selection.

based on synthetic performance data, is presented in section A.5 (p. 223).

The data for the simple CCS-based evaluation has been obtained from the performance experiments described in section 9.2.2 (p. 181), which cover many CCS setups but only consider six SSA algorithms: FR, DM, ODM, and NRM-B combined with three event queues (SimpleEQ, a Heap, and MListRe — see sec. 9.1, p. 178). Portfolio selection is carried out by the genetic algorithm described in section 7.2.2 (p. 143), although the problem is very simple.[6]

The genetic algorithm is configured with a mutation rate of 0.1%, and considers 100 individuals for 20 generations. It has been used to generate both risky and safe portfolios of various sizes, as displayed in table 9.1 (p. 190). Besides speeding up the convergence of the adaptive simulation runner, the constructed portfolios also allow to rank and analyze the considered algorithms.

Firstly, ODM is included in *all* selected portfolios, regardless of portfolio size and the value of λ. It performs best overall and is hence the most promising implementation.[7] For two algorithms, safe and risky portfolios differ: while the risky variant contains DM, the safe one includes NRM with SimpleEQ instead. From this, one can infer that DM outperforms NRM(SimpleEQ) for a *relevant* set of problems, i.e., problems for which ODM is not the best option. Furthermore, the result shows that NRM(SimpleEQ) outperforms DM in terms of average performance (since it is included in the safe portfolios). When selecting exactly three algorithms, a risky portfolios construction results in a combination of ODM, DM, and any other algorithm. This shows that no other algorithm is able to outperform *both* DM and ODM on any given problem. Since $\lambda = 1$, only peak performance per problem is considered in equation 7.6 (p. 144): the first portfolio that contains both ODM and DM has a maximal fitness value and will be selected. To illustrate this point, table 9.1 also gives portfolios for $\lambda = 0.99$ whenever this effect occurs. While portfolios selected with $\lambda = 0.99$ are still very risky,

[6] There are only $2^6 - 1 = 63$ non-empty subsets of algorithms — simply enumerating and evaluating all eligible combinations would suffice in this case.

[7] Note that in case of a portfolio size $s = 1$, both summands of equation 7.6 (p. 144) are equal and hence the value of λ is irrelevant — the optimal single-element portfolio contains the algorithm with best average performance.

Size	**Risky** ($\lambda = 1.0$)	**Safe** ($\lambda = 0.0$)
1	ODM	ODM
2	ODM+DM	ODM+NRM(SimpleEQ)
3	ODM+DM+X *[$\lambda = 0.99$: ODM+DM+NRM(SimpleEQ)]*	ODM+NRM(SimpleEQ)+NRM(MListRe)
1 – 2	ODM+DM	ODM
1 – 6	ODM+DM+X *[$\lambda = 0.99$: ODM+DM]*	ODM

Table 9.1.: Portfolios selected by the genetic algorithm (see sec. 7.2.2, p. 143). It uses the performance data obtained as described in section 9.2.2 (p. 181). Since the objective function neglects the portfolio size for $\lambda = 1$ (eq. 7.6, p. 144), it selects arbitrary algorithms in case all relevant algorithms have already been included; these are denoted by X. To get the smallest risky portfolio, the selections in question have been repeated with $\lambda = 0.99$ — their results are given in square brackets. All selection tasks have been replicated five times, as the genetic algorithm implements a stochastic search.

they give a minimal penalty to those portfolios containing algorithms which never outperform the others.[8] The last two rows in table 9.1 give results for variable-size portfolios. The safe portfolios are little surprising, as the algorithm with the best average performance always maximizes the fitness function in case $\lambda = 0$. The risky portfolios show that both ODM and DM should be included in case adaptation works, but that no other algorithm is required. All the conclusions drawn from table 9.1 may help a performance analyst or developer to decide upon the next steps, e.g., which algorithm deserves most attention. The above analysis matches the results displayed in figure 9.4 (p. 185), but yields additional insights. For example, it is not clear from figure 9.4 that NRM(SimpleEQ) outperforms DM when it comes to *average* performance (as discussed above).

Besides their use for performance analysis, the selected portfolios have to prove their positive effect on the convergence speed of adaptive replication. To show this, the CCS benchmark experiment described before (see fig. 9.7, p. 189) is repeated with some of the multi-armed bandit policies. They are now restricted to algorithms from the best non-trivial risky portfolio, which just consists of two algorithms: ODM and DM (see tab. 9.1). Since this should particularly speed up simulation experiments which include only few replications, the number of replications is reduced from 500 to 50. Figure 9.8 (p. 191) shows the impact of portfolio selection on convergence speed. While being restricted to a portfolio is beneficial for the performance of all considered policies in the given scenario, note that the policies are *not* guaranteed to reach the optimum anymore. If the portfolio does not contain the DM — e.g., in case a safe portfolio of size two is chosen (see tab. 9.1, p. 190) — even the best choice from the portfolio is sub-optimal. The improvement in convergence speed is hence traded with (potentially) inferior peak performance and the efforts to collect the performance data required for portfolio selection.

Summary

The results presented in this section show that adaptive replication can be used effectively in practice, given a number of options that is (much) smaller than the number of required replications. If this is not the case, prior portfolio selection helps to reduce the number of options. The results show that the performance of the adaptive simulation runner strongly depends on the policy that is used, but also that even bad-performing policies do at least as good as a random selection. Previous experiments presented in [72] came to similar conclusions and investigated the performance of the adaptive simulation runner on a larger set of benchmark models, and also with a larger set of options (48 options for simulating a model defined in stochastic π-calculus [200]).

Finally, the failure detection mechanism presented in section 8.1.2 (p. 162) is seldom activated and its execution speed is negligible for the problems currently encountered in JAMES II (regarding the

[8] Consequently, λ should not be set to 1 in any practical setting, as this basically assumes an optimal algorithm selection right from the beginning of the replication, i.e., there is not even a small overhead due to exploration.

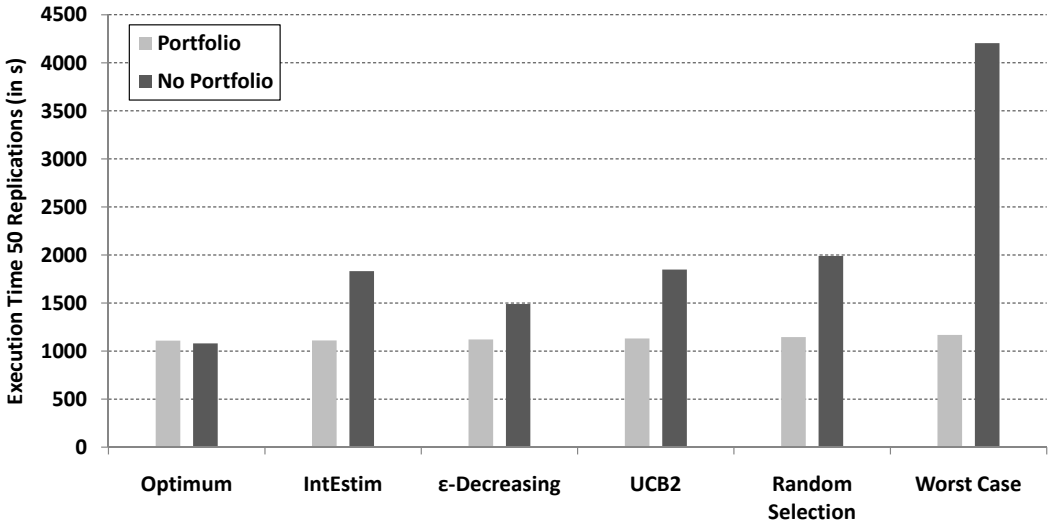

Figure 9.8.: Prior portfolio selection reduces the execution time of the best-performing policy (IntEstim, see fig. 9.7, p. 189) by almost 40%. Executing 50 replications without a portfolio takes much longer for the other policies as well, due to increased exploration overhead. All policies still outperform RandomSelection, but only little can be gained by adaptation here. This is because both portfolio algorithms (ODM and DM) perform rather similar, which is illustrated by the strong reduction of worst-case execution time (only ≈ 5.3 % below the optimum). These circumstances cannot be guaranteed in general, so that using an adaptation policy is still advisable. Since DM is the optimal choice for all six setups contained in the experiment, both old and new optimum converge to the same value; their difference here is due to stochasticity.

number of plug-ins involved per simulation run). Therefore, no quantitative results are presented here; the interplay between the adaptive simulation runner and failure detection has been shown to work by unit tests.

9.2.4. Selector Generation

In order to come up with a trustworthy selection mapping to be established for the SSAs in general, more performance data needs to be gathered. Additionally, the selectors generated by the SPDM (see ch. 6, p. 115) need to be validated against real-world models, e.g., from the biomodels database [64]. Since this is beyond the scope of this thesis, the following experiments are again restricted to the CCS benchmark model and the data obtained from scenario three (described in sec. 9.2.2, p. 181).

Tested Selectors

As discussed in chapter 6 (p. 115), choosing a suitable form of selection mapping is hard, even for the relatively small amount of performance data considered here — there are only 150 simulation problems (see sec. 9.2.2, p. 181). Similar to the experiments presented in section 8.2 (p. 166), the selector generation mechanisms described in section 6.2.1 (p. 121) are applied to the performance data and their prediction errors are compared with each other. As in section 8.2 (p. 166), the performance metric to be predicted is throughput, i.e., the progress in simulation time per unit of wall-clock time. Four features are extracted from each model: the number of species (for CCS: N), the number of reactions (for CCS: $N \cdot r$), the average number of reactants and products (for CCS: k), and the average initial population per species (for CCS: $\bar{X} = X_1 = \ldots = X_N$). While the initial size of species populations is not constant over time, it may still yield important insights. For example, if the population sizes strongly impact SSA performance, this could motivate (and facilitate) the development of an adaptive scheme that switches SSA implementations at runtime. Furthermore, note that there are many other

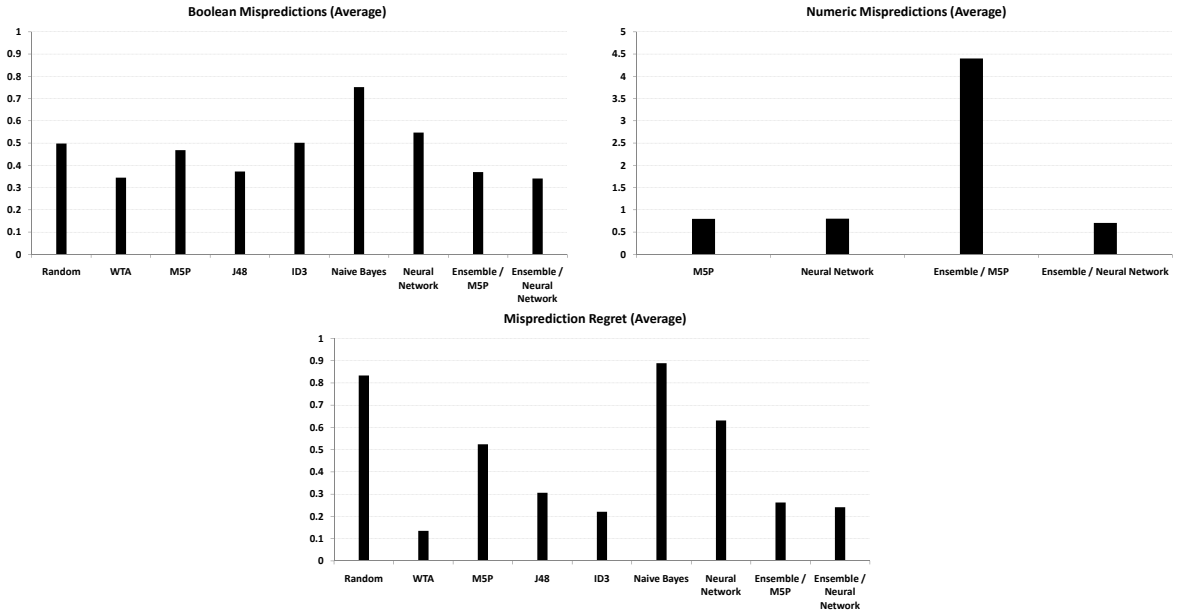

Figure 9.9.: Selector performance estimated by the measures described in section 6.2.2 (p. 125), which are all relative. Boolean mispredictions (upper left) gives the error percentage of deciding which of two configurations will be the fastest for a feature combination from the test set. Numeric mispredictions (upper right) quantify this error by giving the average difference to the true performance in percent. Since classifying methods like decision trees (J48, ID3) do not give quantitative predictions, this measure only applies to a subset of methods. Misprediction regret (bottom) plots the performance overhead with respect to an optimal selection (i.e., S^*) in percent.

features of SSA models that are not considered in this illustrative example, e.g., the distribution of reaction propensities, the ratios between reaction constants, and so on.

Selector generation is carried out by Weka, MLJ, and Joone (see sec. 6.2.1). Weka provides the decision tree approach J48 and the model tree approach M5P (see sec. 6.2.1, p. 121). J48 is configured to distinguish 20 different performance classes, M5P uses default settings. MLJ provides the decision tree approach ID3 and a Naïve Bayes learner (see sec. 6.2.1, p. 121). ID3 is configured like J48, i.e., it also distinguishes between 20 performance classes. The same holds for Naïve Bayes. Finally, Joone provides the construction of neural networks (see sec. 6.2.1, p. 122). It is configured to use resilient back-propagation [216, p. 70] with five hidden layers of 20 neurons each. The neurons are configured to have a linear transfer function and are connected with all neurons of the previous layer. The learner is configured to execute 100 training cycles with a learning rate of 1.0 and a momentum of 0.5. The learning rate controls the extent of a weight change, while the momentum controls the impact of former weight changes on the current change [216, p. 69]. Furthermore, the schemes that implement regression (M5P and neural networks) have been combined with the `EnsembleSelectorGenerator` (see sec. 6.2.3, p. 128), which trains an individual predictor for each of the six SSA algorithms and then decides by querying the individual predictors and comparing their (quantitative) performance predictions.

Selector Performance Measures

Figure 9.9 shows the three performance measures for selectors that are described in section 6.2.2 (p. 125). They result from 10 replications of bootstrapping (see sec. 6.2.2, p. 124).[9] Firstly, the way the Naïve Bayes classifier is used here seems to be ineffective. It fails to decide for many pairs of

[9]The results obtained by a 15-fold cross-validation, also replicated 10 times, are similar.

configurations, so that it often falls back to predicting equal performance for both configurations that are compared. Unless both configurations really yield exactly the same throughput, such indecisiveness will be counted as an error (according to eq. 6.4, p. 126). This explains why Naïve Bayes performs *worse* than a random selection when it comes to the boolean performance measure (fig. 9.9, upper left plot). The misprediction regret (fig. 9.9, bottom plot) shows that Naïve Bayes is also unable to outperform a random selection when it comes to a hypothetical selection of the most suitable simulation algorithm for a problem. Hence a Naïve Bayes approach seems unsuitable to cope with this kind of problem, probably because the data set is too small (150 problems only) and the basic assumption of the classifier is not met, i.e., the test set did not contain data on problems with exactly the same features as those in the training set (see sec. 6.2.1, p. 121).

All other kinds of selectors are able to outperform a random choice in terms of boolean mispredictions, except for using a single neural network, which performs only slightly worse. This means most selectors are effective in deciding which of two algorithms will outperform the other on a certain problem — yet none of them delivered good performance. The best selectors in terms of boolean mispredictions are generated by the `EnsembleSelectorGenerator` when using dedicated neural networks to predict the performance of each algorithm individually (see sec. 6.2.3, p. 128). However, even those perform only slightly better than the winner-takes-all approach, which just ranks each algorithm by overall performance and sticks to this ranking without considering any problem features (see sec. 6.2.3, p. 128). In other words, none of the selector generation mechanisms is able to come up with a selector that drastically improves the chance of guessing which of two algorithms is faster by taking into account the extracted problem features.

There may be several reasons for this result. Firstly, the collected data may be insufficient: the extracted features may not allow to predict algorithm performance (this was ruled out for the effectiveness experiment in sec. 8.2, p. 166), also the data set may still be too small to train good selectors. Secondly, the forms of selection mappings, as provided by the selector generators available in the SPDM so far, may not be appropriate. For example, M5P assumes linear relationships between problem features and performance. Given M5P's performance, particularly in terms of numeric performance prediction when used by the `EnsembleSelectorGenerator` (fig. 9.9, p. 192, upper right plot), at least some of the algorithms seem to depend on the model features in a non-linear way. This also explains why neural networks do comparably well when trained by the `EnsembleSelectorGenerator`: they are the only SPDM selectors that allow quantitative predictions and can cope with nonlinearity.

Finally, it should be noted that the winner-takes-all approach does remarkably well, particularly when considering the most important performance measure, misprediction regret (fig. 9.9, p. 192, bottom plot): even such a simple way of algorithm selection only introduces $\approx 13\%$ overhead w.r.t. the optimum in this scenario, without considering any problem features. While no learned selector was able to achieve better or similar overhead, ID3 and ensembles of neural networks achieve an overhead of only $\approx 22\%$ and ≈ 24 %, respectively. Although they do not outperform the winner-takes-all strategy here, they outperform a random selection by a good margin (it introduces ≈ 83 % overhead). Since all selectors (apart from Naïve Bayes) outperform a random selection with respect to misprediction regret, they all can be regarded as average-effective for the considered scenario (see def. 2.1.5, p. 17). However, none outperforms the winner-takes-all strategy — so none of them is adaptive-effective (see def. 2.1.6, p. 18).

Note that the adaptive gain (see def. 2.1.8, p. 19), which basically defines an upper bound on the speed-up achievable by replacing the best constant selection mapping S^*_C with one that considers problem features, is only about 0.15. In other words, a feature-based algorithm selection may outperform the best-performing selector that always picks the same algorithm by at most 15% (for the given problems). This suggests that a simple selection mechanism suffices in this specific scenario — its overhead is rather small.

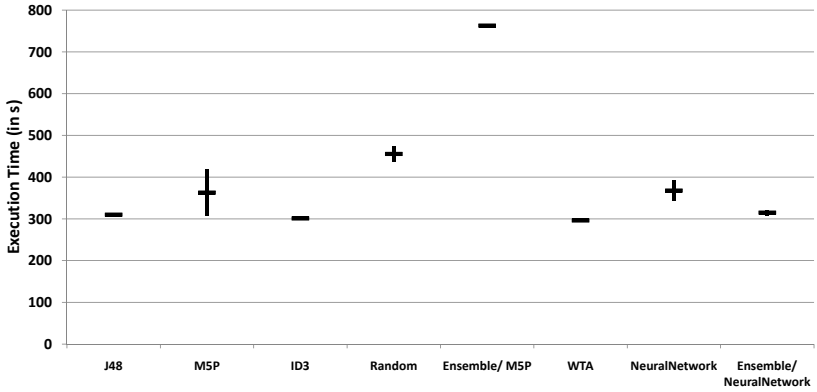

Figure 9.10.: Real-world performance of selectors generated by different mechanisms, executed on nine formerly unseen CCS setups. Each experiment has been replicated ten times. Horizontal bars denote the average execution time, vertical bars display the standard deviation in both directions (i.e., average execution time $\pm\sigma$).

Real-World Performance

To test the real-world effectiveness of the selector generators, they have been used to create selectors from *all* data gathered in scenario three of section 9.2.2 (p. 181) — i.e., no data has been left out for testing or validation. Instead, the real-world performance of the selectors is evaluated by applying them to CCS setups that are *different* from the 150 CCS setups that have been considered for selector generation. The initial number of particles has been set to $X_i = 10^5$, instead of being from $\{2000, 5000, 8000\}$ (see sec. 9.2.2, p. 181). Similar to the setups for evaluating adaptive replication in section 9.2.3 (p. 186), the number of species (N) and the factor for the number of reactions (r) are varied. Nine setups are considered — all combinations with $N \in \{3, 4, 5\}$ and $r \in \{1, 2, 3\}$. Still, the parameter deviations are relatively small, so the results merely illustrate performance gains in case the simulation problems considered for prior performance analysis are rather representative.

Each CCS setup is replicated three times,[10] so there are $9 \cdot 3 = 27$ runs to be executed with each selector. The simulation end time is set to 50 s. Figure 9.10 shows the results of these experiments: except for the ensemble of M5P predictors, which performs very bad, all learned selectors outperform a random selection. Similar to the performance metrics evaluated before (see fig. 9.9, p. 192), however, none is able to outperform the simple winner-takes-all policy on this small set of nine unseen problems — yet, the performance of the decision trees (ID3 and J48) is almost identical, with an overhead of only 1.6% and 4.7%, respectively.

The M5P ensemble performs so bad because it selects the NRM in combination with a Heap for all nine problems, which is not a good choice here. This is caused by a misleading performance model for this particular runtime configuration, i.e., the performance of NRM(Heap) is predicted to be much better than it actually is. Potential problems with the M5P ensemble are already suggested by the large numeric prediction error of this selector, as depicted in figure 9.9 (p. 192). It illustrates why more than one performance measure is needed: all selectors that perform well with respect to numeric misprediction *and* misprediction regret are able to outperform a random selection. The performance measures shown in figure 9.9 (p. 192) should be regarded as necessary conditions[11] for good selector performance on unseen problems.

[10] Replications are used here to reduce the runtime variation of experiments with the random selector (see sec. 6.2.3, p. 128), which may pick different algorithms for each replication.

[11] Clearly, none of these conditions can be sufficient, due to the problem of inductive knowledge (see sec. 1.4, p. 8).

Summary

The evaluation of the SPDM in this context is rather brief, as the circumstances do not allow any significant improvement in execution time by generating a non-trivial selector. While this result seems unsatisfying from an algorithm selection viewpoint, it would be good news in practice: this particular algorithm selection problem is not hard to solve, since even a simple selection strategy achieves good performance. This situation will be different for other sets of algorithms, other benchmark problems, and other application domains. Besides that, one can only be confident in relying on the winner-takes-all strategy *after* its performance has been evaluated and compared to other approaches by the SPDM. The SPDM is hence quintessential for solving the ASP in JAMES II, even though it is not necessarily able to come up with adaptive-effective selectors. That the creation of such selectors is possible in principle has been shown in section 8.2 (p. 166), and will also be illustrated in the second case study (sec. 10.2.4, p. 203).

9.3. Summary

This chapter shows how to apply the simulation algorithm selection methods from the second part of this thesis to the simulation of chemical reaction networks. The experiments brought about the following main results:

- An efficient collection of performance data is crucial to the overall selection process. Performance experimentation must be carried out carefully and needs to consider simulation-specific aspects. For example, a calibration of the simulation end times allows to automate experimentation and may speed up experiments by almost one order of magnitude (see fig. 9.2, p. 183). At the same time, it also improves the reliability of the collected data (see fig. 9.4, p. 185). Additional tools can be used by developers to exploit performance data during simulator development, as briefly illustrated by the usage of the algorithmic change evaluator (see fig. 9.5, p. 185).

- Adaptive replication is shown to be effective for stochastic simulation studies (see fig. 9.7, p. 189). The pre-selection of a simulation algorithm portfolio makes the approach applicable to experiments that require relatively few replications. Combining both techniques allowed to achieve near-optimal performance under the given circumstances (see fig. 9.8, p. 191).

- Almost all selectors generated by the SPDM outperform a random selection considerably, both in terms of predicted performance (see fig. 9.9, p. 192) and within a small real-world test setup (see fig. 9.10, p. 194). However, the simple strategy of choosing the algorithm that is fastest on average — the ODM — suffices in this scenario to achieve near-optimal performance. More complex selectors that take problem features into account could achieve a similar performance (e.g., the ID3-based selector), but they could not outperform it. This situation, however, wholly depends on the problems that are considered and the available algorithms, in other words the specifics of the simulation algorithm selection problem. The general effectiveness of the simulation algorithm selection framework in constructing adaptive-efficient selectors has already been shown in section 8.2 (p. 166)

10. Case Study II: Parallel Discrete-Event Simulation

> *While the water is heating, you have a choice of what to do — just wait, or do other tasks in that time such as starting the toast (another asynchronous task) or fetching the newspaper, while remaining aware that your attention will soon be needed by the teakettle. The manufacturers of teakettles and toasters know their products are often used in an asynchronous manner, so they raise an audible signal when they complete their task. Finding the right balance of sequentiality and asynchrony is often a characteristic of efficient people — and the same is true of programs.*
>
> Goetz et al. [111, p. 2]

While the case study on SSAs (ch. 9, p. 177) ought to show the benefits and limitations of the developed methods for simulation algorithm selection, the following study shall mainly illustrate that these methods are not restricted to SSAs. They rather provide a general toolkit to analyze simulation algorithm performance and thereby allow us to solve the algorithm selection problem in a simulation context. To show the generality of the methods does not require a full evaluation of every technique, as has been done in chapter 9 (p. 177). The study is hence focused on the most relevant methods.

As another contrast to the first case study, the algorithms investigated here are not application-specific. Parallel and distributed discrete-event simulation (PDES) lends itself well to many application domains where large and computation-intensive models are considered, e.g., the simulation of computer networks or traffic phenomena (see sec. 1.3.2, p. 6). The algorithms provided by JAMES II enable a parallel execution in that they use multi-threading to exploit the parallel computing capabilities of a single machine, e.g., one with a multi-core CPU. They do not yet allow a *distributed* simulation, so that this study deals with parallel — but not distributed — discrete-event simulation.

10.1. Algorithms under Consideration

The implementations of the algorithms that are evaluated here have already been presented in [321]. All of them are fundamental in that their basic principles are widely applied today, albeit in combination with various enhancements (e.g., see [90] for details).

They operate on a model that can be regarded as a labeled directed graph: the nodes are logical processes (LPs), i.e., model entities that execute events and produce new ones by doing so. Logical processes are connected with each other by directed edges, so that each process may schedule events for its neighbors (w.r.t. outgoing edges) and may receive new events over its incoming edges. Oftentimes, processes can guarantee a certain temporal delay between the local execution of an event and the time stamps of events that are scheduled to a certain neighbor. This delay is called *lookahead*, and it can be regarded as an edge label in the directed graph of LPs. The notion of lookahead is important to grasp the limitations of the simulation algorithms discussed in the following.

Consider the classical example (see [90, p. 52 et sqq.]) of an air traffic simulation, where each logical process is an airport and the graph edges denote flight routes. One of the LPs, RLG, may represent the airport Rostock (Laage). It has an outgoing edge to the LP CGN, which represents the Cologne airport. A civil aircraft typically takes an hour for this trip, so the lookahead for the edge from RLG to CGN could be set to $l = 45$ minutes simulation time. Regardless of the state RLG is in, it can *guarantee* CGN that *all* events caused by the currently executed event, e.g., a landing

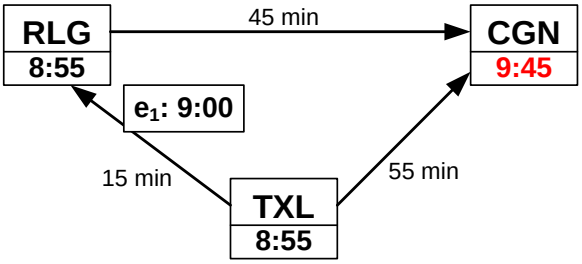

Figure 10.1.: Lookahead example: since RLG receives (and executes) an event from Berlin/Tegel (TXL) at 9:00 o'clock, it can now *guarantee* the Cologne LP (CGN) that the latter will not receive any events from RLG with time stamps less than 9:45. One way to deliver such guarantees is by using null messages (see p. 199).

aircraft at t = 9:00 o'clock, will only produce events for CGN with time stamps larger or equal to $t + l$. In this case, $t + l$ = 9:45 in simulation time. Figure 10.1 illustrates the notion of a lookahead. Generally speaking, a high lookahead suggests a large degree of model-inherent parallelism, as events occurring at different processes only impact each other at some later time. Note, however, that there are also *zero lookahead* models, in which LPs cannot give such guarantees. An example for such a kind of model are the chemical reaction networks discussed in section 9 (p. 177): reaction times are exponentially distributed and hence time delays may be arbitrarily close to 0. The efficient execution of such kinds of models is very challenging (e.g., [160]). Finally, a lookahead is not necessarily fixed during the execution of a simulation run — however, a fixed lookahead is assumed in the following.

Note that the following algorithm descriptions are *very* brief, for the sake of clarity. This, however, leaves out the discussion of some performance-relevant aspects — for example the computation of the *global virtual time (GVT)*, i.e., the global minimum of the local virtual times (see sec. 1.3.2, p. 6) and hence the current simulation time, or mechanisms to cope with transient messages. Some more details are given in [321]; more knowledge on the algorithms is not necessary to judge the performance of the simulation algorithm selection methods. Approaches to predict and analyze the performance of PDES algorithms have already been discussed in section 3.3 (p. 70).

Sequential Simulator (S)

The sequential simulator works as already outlined in section 1.3.2 (p. 6): events are stored to an event queue, which orders them by their time stamps. The sequential simulator merely retrieves the event with the smallest time stamp from the event queue, invokes the corresponding LP to execute it, and adds the newly generated events to the central event queue (see fig. 1.3, p. 7).

Barrier Synchronization (BS)

Barrier synchronization is a conservative approach to PDES, i.e., it restricts each thread to only execute those events that are *safe* in the sense that they do not violate the local causality constraint (see sec. 1.3.2, p. 6). The basic idea is straightforward: at first, each LP_i calculates up to which time stamp t_i it is safe to process the events in its event queue. This calculation involves the current simulation time, lookahead values, and the time stamps of the events in the queue. A central thread gathers these time stamps and calculates the minimum $t_{min} = \min_i t_i$, which is returned to all LPs as the new time limit up to which event execution is safe. Now, every LP executes all events up to t_{min}, sends new events out to their destination LPs, recalculates its local t_i, and sends it to the central thread. Afterwards, it synchronizes with a barrier, i.e., it waits for all other LPs to finish this procedure as well.

After all processes have synchronized with the barrier, execution proceeds in that a new t_{min} is calculated, and again all LPs execute their safe events with time stamps less or equal to t_{min} in

parallel. Here, a central barrier is chosen [90, p. 66], namely the cyclic[1] barrier implementation of the Java concurrency API (see e.g., [111]). A more detailed description of this simple synchronization protocol can be found in [90, p. 74–75].

Null Messages (NM)

The *null message protocol* [90, p. 54 et sqq.] — also named after its inventors, i.e., Chandy/Misra/Bryant — is a conservative synchronization mechanism as well. In contrast to barrier synchronization, it avoids a central synchronization step by letting each LP consider the lookahead and local virtual times of LPs with incoming edges. If another LP with an incoming edge has a local virtual time of x and the edge has a lookahead of l, this means it will not schedule any events to the given LP with time stamps less than $x + l$. Hence, if an LP has an event in the queue with a time stamp *less* than the minimal $x + l$ from any LP with an incoming edge, this event is safe to be processed. Unfortunately, this simple protocol is prone to deadlocks, i.e., situations where LPs wait for each other and the simulation comes to a halt (see sec. 1.3.2, p. 6).

These situations can be avoided by using so-called *null messages*. After processing an event, an LP just sends messages to *all* LPs to which it has an outgoing edge. These messages only contain its new local virtual time plus the edge-specific lookahead, and hence represent a guarantee: the given LP guarantees that it will not send an event with a smaller time stamp over this outgoing edge in the future. Since these guarantee messages do not contain simulation events, they are called null messages. Not that there is also a variant of the null message protocol (see discussion in [90, p. 57–58]) where the LPs request null messages on-demand, in order to reduce the overhead due to messaging. Furthermore, the null message protocol is only able to avoid deadlocks if the LP graph does not contain any cycle with zero lookahead edges, and is hence restricted to a sub-class of models.

Time Warp (TW)

Time Warp [90, p. 97 et sqq.] is a well-known and popular PDES synchronization protocol. It is the prototypical optimistic synchronization mechanism, i.e., it risks to temporarily violate the local causality constraint in order to better exploit the model-inherent parallelism (see sec. 1.3.2, p. 6).

Each LP executes all events in its local queue as fast as possible. If a violation of the local causality constraint is detected, i.e., a straggler event with a time stamp less than the local virtual time is received (see sec. 1.3.2, p. 6), all events that have been processed in the meantime are rolled back. This does not only mean to restore the prior state of the LP, but also to invalidate all *new* events that were generated by the events to be rolled back. This is done by keeping book of all messages that have scheduled events to other LPs. If such a message has been caused by a now-invalid event, the LP sends a corresponding *anti-message* to its destination. In analogy to a matter/anti-matter reaction, an anti-message shall ultimately annihilate the original message, so that both can be safely deleted if found within an LP's local event queue. If the event contained in the original message has already been processed, the anti-message causes the other LP to roll back as well. This effect is known as a rollback cascade, as it may involve many LPs and hence slow down the overall simulation significantly. The given implementation sends out anti-messages in an aggressive manner, i.e., as soon as a rollback occurs (other alternatives are discussed in [90]).

Keeping book of all messages and prior LP states[2] consumes considerable amounts of memory. Consequently, book-keeping should be restricted to those messages and states that actually need saving, i.e., those that *could* be rolled back. Therefore, the Time Warp protocol repeatedly initiates the calculation of the global virtual time, which is the minimum of the local virtual times of all LPs on the one hand, and the minimal time-stamp of all transient messages on the other hand. By definition, no LP can be rolled back any further than to the GVT, so that all old states and messages with smaller time stamps can be safely removed from memory. This procedure is called *fossil collection*

[1] It is cyclic in the sense of being re-usable for many cycles of repeated synchronization.

[2] Alternatively, the prior LP state could also be reverse-computed. For this, each event needs to define an *invertible* transformation of the LP state, which may be non-trivial.

(e.g., [90, p. 110]), and is implemented in batch mode here: as soon as a the GVT is updated, all old states and messages are dismissed.[3] In the given implementation, GVT computation is invoked periodically after sleeping 1 s and letting the simulation proceed in the meantime.

Breathing Time Buckets (BTB)

The *breathing time buckets (BTB)* protocol (e.g., [90, p. 157–159]) is also an optimistic synchronization scheme, but it constrains the optimistic execution and avoids rollback cascades. BTB works in three main phases:

1. Each LP processes events from its local event queue, until the time stamp of a newly generated event is larger than the time stamp of the next event to be processed. All newly generated events are collected, but *not* sent to any other LP.
2. The minimum time stamp of all newly generated events is computed. This is the global virtual time, as all events with a smaller time stamp have been safe to execute in retrospect — no new event can invalidate them, and all old ones have already arrived at the receiving LPs. There are no transient messages.
3. The new GVT is sent to all LPs, which in turn send all newly generated events that are safe to process, i.e., with a time stamp less or equal to GVT. All processed (and newly generated) events with larger time stamps may or may not be valid and could later be rolled back. This depends on the safe events each LP now receives from the other LPs. After each LP has rolled back those optimistically executed events that are invalidated by the received events, the protocol starts over.

Event Queues

As with some of the SSA algorithms in chapter 9 (sec. 9.1, p. 179), both the sequential simulator and BTB can be customized with JAMES II event queues. The simple event queue (a sorted list) and the Heap-based event queue are used here (see sec. 9.1, p. 179).

Partitioning and Load Balancing

Since all LPs are executed by single threads on a multi-core machine, no partitioning is necessary. Load balancing is handled by the runtime environment, i.e., the JVM and the operating system.

10.2. Experimental Evaluation

This section follows the same steps as the case study presented in the previous chapter (see sec. 9.2, p. 180): after detailing the overall setup (sec. 10.2.1), algorithm performance is explored (sec. 10.2.2) and both adaptive replication (sec. 10.2.3) and the application of the SPDM (sec. 10.2.4) are evaluated.

10.2.1. Setup

The benchmark model used throughout the experiments is the PHOLD model, as described in section 7.3.1 (p. 149). It simply consists of a network of LPs that, upon receiving a new event, generate a new event with a larger time stamp and send it to a random neighbor. PHOLD is configurable in many ways. For the following experiments, the model is altered in three ways:

[3] Also note that the fossil management of the given implementation exhibits some problems: there are some rare cases — less than ten times during 10 s of wall-clock time, i.e., during the execution of many thousand events — when a cascading rollback reaches an LP where its prior state *cannot* be found anymore. Due to its rareness, the effect of this erroneous behavior on the overall execution time should not be too strong — and since LPs in PHOLD are essentially stateless, this problem should neither impact the simulation run time in general.

- *Network topology*: As the results in [321, p. 1176] indicate, the topology of the LP network may significantly impact the performance of the synchronization scheme. The same topologies as tested in [321] are used here: a fully connected graph (`FULL`), i.e., each LP has all other LPs as neighbors, a toroidal grid (`GRID`), i.e., each LP has four neighbors, and a ring of LPs (`RING`), i.e., each LP has just two neighbors.
- *Size*: The number of LPs in the network. To facilitate the construction of a grid topology, this number is restricted to square numbers, i.e., $2^2 = 4$, $3^2 = 9$, $4^2 = 16$, and so on.
- *Number of events*: The number of events that are initially created and scheduled to random LPs. Since each event will cause the generation of a new event, i.e., the amount of events to be processed stays constant, this directly affects the overall computational load imposed by the model. Each event only generates a small synthetic load,[4] but adding an event to the initial PHOLD state will cause the generation of *many* new events over the course of the simulation. Also note that a growing number of events increases the model-inherent parallelism; all initial events are independent of each other and can be processed in parallel.

New events are scheduled at time $t + \delta + X$, where t is the time stamp of the previous event, $\delta = 1$ serves as the non-zero lookahead (to make the null message protocol applicable, see sec. 10.1, p. 199), and X is an exponentially distributed random variable with $\lambda = \frac{1}{4}$. Therefore, the expected value of the new event's time stamp is $t + \delta + \lambda^{-1} = t + 5$. The technical setup, i.e., the machine the experiments are executed on, is the same as before (see sec. 9.2.1, p. 181).

10.2.2. Simulation Space Exploration

To explore the performance of the simulation algorithms and record the data required for selector generation (detailed in sec. 10.2.4, p. 203), the three PHOLD parameters described above are varied as follows:

- The number of LPs is chosen from $\{4, 16, 36, 64, 100, 144, 196, 256\}$.
- The number of events is defined as a *factor* of the number of LPs. It is chosen from $\{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$. For example, if the number of LPs is 100 and the factor is $\frac{3}{4}$, then $\frac{3}{4} \cdot 100 = 75$ events exist in the model at any point in simulation time.
- The three topologies: `FULL`, `GRID`, and `RING`.

All parameter combinations amount to $8 \cdot 4 \cdot 3 = 96$ setups. Since the effectiveness of the simulation end time calibration and the adaptive simulation runner has already been shown (scenario three, sec. 9.2.2, p. 181), they are again combined to conduct this experiment. The given simulation algorithms, however, are not necessarily sequential anymore; instead they may use *all* eight cores that are available on the workstation. As multi-threading is likely to introduce additional noise to the run times, the simulation end time calibration is configured slightly differently here: t_{opt}^{wct} is set to 10 s, t_{max}^{wct} to 50 s, t_{min}^{sim} to 100 time units,[5] and t_{max}^{sim} to 10^5 time units. The simulation end times have been adjusted to account for the relatively large delays between events (on average 5 time units, see above). The simulation end time calibrator is configured to use two algorithms, barrier synchronization and BTB with the Heap-based event queue (see sec. 10.1, p. 197). All other components (e.g., the adaptive simulation runner) have been configured identically to scenario three, as described in section 9.2.2 (p. 181).

The overall experiment execution took ≈ 9.83 hours, with only ≈ 18.7 minutes ($\approx 3.2\%$) of the execution time spent for calibration. The calibrated simulation end times are depicted in figure 10.2,

[4] The load is generated by a `for`-loop that calculates a sum.

[5] In contrast to the SSAs, which rely on the rate constants and their unit of time, the time scale of PHOLD is not specified.

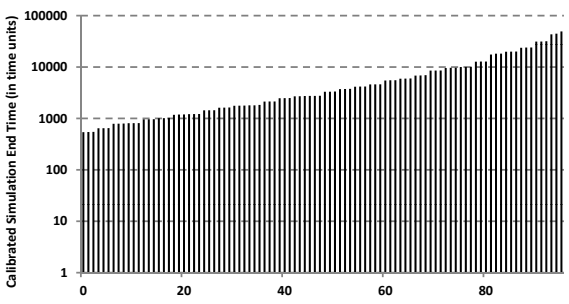

Figure 10.2.: Calibrated simulation end times for the 96 setups of the PHOLD model, in ascending order. Note the logarithmic scale, i.e., the end times span two orders of magnitude.

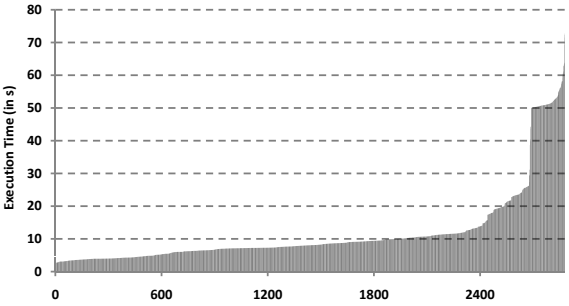

Figure 10.3.: Execution times for PHOLD setups with calibrated simulation end times, in ascending order.

the average end time is 7434.45 time units. Only 386 executions are required for calibration, i.e., ≈ 2 unsuitable end times have been tested per simulation problem (386 divided by 96 setups $\times$ 2 algorithms to execute).

The corresponding wall-clock times for executing the simulation problems with the calibrated end times are shown in figure 10.3. There are 2880 executions on problems with calibrated simulation end time, i.e., 30 executions for each of the 96 problems: 28 are carried out by the adaptive simulation runner (7 algorithms $\times$ 4 replications per algorithm, just as in sec. 9.2.2, p. 181). The other two executions again stem from the last round of the calibration mechanism, where barrier synchronization and BTB with a Heap are already applied to a simulation problem with a suitable simulation end time. The average execution time is ≈ 11.9 s, which is a bit longer than the desired execution time of $t_{opt}^{wct} = 10$ s.

Also note that all execution times ≥ 50 s in figure 10.3 stem from censored runs, i.e., these executions have been aborted after t_{max}^{wct} was reached. Yet, there are some runs that lasted considerably longer, with a maximum of 72.8 s. All these executions are due to a single algorithm, Time Warp, which is occasionally so slow that even a GVT computation takes several seconds (more precisely $72.8 - 50 = 22.8$ s): only after a single GVT computation has been completed, the control of the simulator is given back to JAMES II, in order to decide whether the simulation run should proceed or not. This suggests that the given Time Warp implementation, or at least the way it handles GVT calculation, is implemented in a sub-optimal manner, a claim that is backed up by the fact that over 75% of all Time Warp executions have been censored (154 out of 202) and these executions represent over 80% of all censored executions (154 out of 192). Since the given Time Warp implementation performs so badly, it is excluded from the following experiments.

Figure 10.4 shows the number of times each algorithm came out with the best or worst performance for one of the 96 PHOLD setups. As already discussed, Time Warp is the worst-performing algorithm for almost all setups. The null message protocol, on the other hand, performs best on 52 of 96 simulation problems, and is hence the best-performing PDES algorithm overall.

10.2.3. Adaptive Replication

As in chapter 9 (p. 177), the evaluation of the adaptive simulation runner is done by considering slightly different setups than those that were considered for performance exploration in section 10.2.2 (p. 201). Here, the number of LPs is chosen from $\{25, 49\}$, in combination with all topologies (`FULL`, `GRID`, `RING`); hence each experiment consists of 6 PHOLD setups. The number of events is set fixed to 2.25, i.e., there are more than twice as much events as LPs in the model at any time. The simulation end time is set to 2000 time units. Each setup is replicated only 50 times, as there are only six algorithms to choose from (the ones described in sec. 10.1, p. 197, except for TimeWarp).

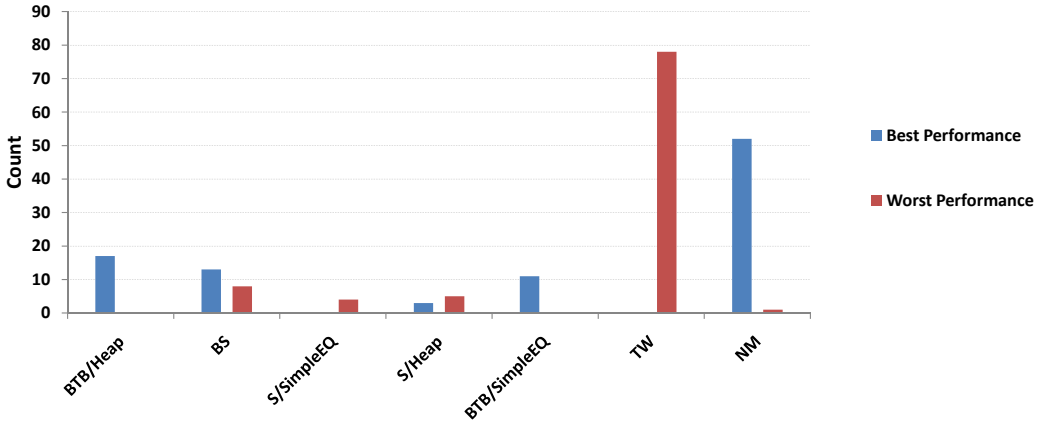

Figure 10.4.: Best/worst performance counts for PDES algorithms. In contrast to the SSA case study (see fig. 9.4, p. 185), where only two algorithms dominated all others, here almost all algorithms outperform the others for at least one simulation problem (except for Time Warp and the sequential simulator using the simple event queue).

To reduce the impact of noise due to multi-threading, only six of the eight available cores have been used for the following experiments, leaving out two cores for the operating system and background tasks. It should be noted that the overall scenario shall, above all, illustrate that the adaptive replication approach presented in section 7.2 (p. 134) indeed works for any kind of stochastic simulation. Apart from that, the given setup makes little sense: all PDES simulators are restricted to the six cores available on the local machine; if multiple replications are required it should always be faster to let instances of a *sequential* simulator run on each core individually. The latter setup does not require any communication between threads during a simulation run, and should almost always outperform PDES simulators that run on a single machine.[6]

Figure 10.5 (p. 204) shows the relative overhead of the multi-armed bandit policies for the most challenging PHOLD setup that was executed. The policies have been configured just as described in section 9.2.3 (p. 186). The best-performing policies already reduce the relative overhead by 50% (w.r.t. RandomSelection) after only 10 replications. After 50 replications, the overhead is only $\approx 16\%$, i.e., using one of these policies is only $\approx 16\%$ slower than knowing the optimal algorithm right from the beginning.

The good performance of the policies is also illustrated in figure 10.6 (p. 205): all of them yield a speed-up with respect to selecting algorithms randomly. The best-performing policy is UCB1-Tuned, which just requires $\approx 20\%$ more time than the optimum to finish the whole experiment. It yields a speed-up of ≈ 2.2 (w.r.t. RandomSelection).

Similar to the results described in section 9.2.3 (p. 187), the performances of policies that belong to the same family (see sec. 7.2.1, p. 136) are clustered together. In contrast to the SSA-study, however, the *order* of the policy families is now different: while ϵ-policies performed better than UCB policies on the SSAs (see fig. 9.7, p. 189), they now perform worse. It has yet to be shown why this is the case; e.g., the number of options could play a crucial role. However, the IntEstim policies are performing well in *both* studies, so they seem to be the most robust policies and should be used per default.

10.2.4. Selector Generation

Finally, the ability of the SPDM to generate suitable selectors for the PDES domain shall be evaluated. The experimental setup is very much the same as the one for the SSAs (cf. sec. 9.2.4, p. 191), i.e.,

[6] An exception may occur if there are more cores than required replications — but in that case adaptive replication needs to be complemented with a more advanced scheduling scheme.

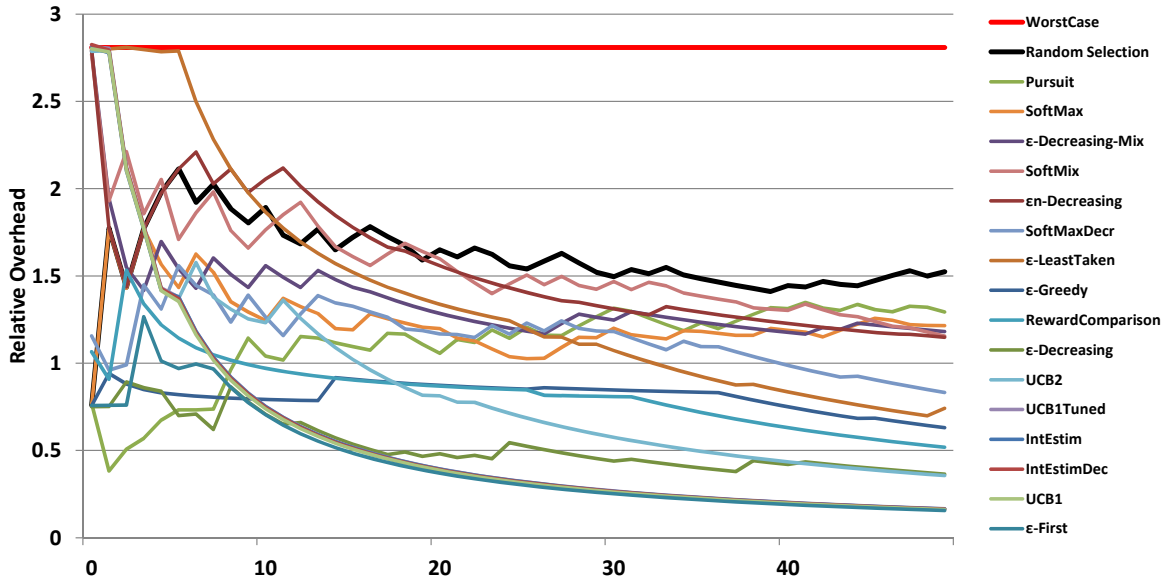

Figure 10.5.: Relative overhead of policies on a PHOLD setup: the same plot as depicted in figure 9.6 (p. 188). Again, the legend on the right side is ordered by the policies' relative overhead after 50 replications. All policies are able to outperform a random selection, though their convergence speeds differ significantly. As before, worst-case and optimum performance have been estimated (see sec. 9.2.3, p. 187).

the same selector generators are used: random selectors, winner-takes-all selectors (both sec. 6.2.3, p. 128), M5P selectors, J48 selectors (both sec. 6.2.1, p. 121), ID3 selectors, Naïve Bayes selectors (both sec. 6.2.1, p. 121), and neural network selectors (sec. 6.2.1, p. 122). As in section 9.2.4 (p. 191), the `EnsembleSelectorGenerator` (sec. 6.2.3, p. 128) is used as well, in combination with M5P and neural networks. Afterwards, the real-world performance of the generated selectors is investigated. This is done by deploying the selectors to the `AlgoSelectionRegistry` and applying them to a small experiment, which consists of formerly unseen simulation problems. The only difference to the study in section 9.2.4 (p. 191) is in the model features that are considered for algorithm selection. Here, four model features are extracted from the LP network to be simulated:

- The number of LPs, n_{LP}.
- The average out-degree $\bar{o}$, i.e., the average number of LPs to which an LP may schedule an event. For the `FULL` topology (see sec. 10.2.1, p. 200), $\bar{o} = n_{LP} - 1$, for the `GRID` topology $\bar{o} = 4.0$, and for the `RING` topology $\bar{o} = 2.0$. The PHOLD topologies used in the following are very regular, i.e., *all* LPs have the same number of neighbors. This, however, is not likely to hold for other models — for these, additional features that characterize the *distribution* of the out-degrees need to be defined as well, e.g., the median, the standard deviation, and so on.
- The average lookahead $\bar{l}$ per edge. As discussed in section 10.2.1 (p. 200), the lookahead is globally set to 1.0 — so this feature will not yield any insight for the learners, as it is always the same. In practice, however, a large-enough lookahead could lead to favoring conservative instead of optimistic synchronization schemes. As with $\bar{o}$, the *distribution* of lookaheads should be considered for a more complete picture, not just the average.
- The average number of initial events per LP in the system, $\bar{e}$. This is constant in PHOLD, but usually not in other models. Similar to the population size feature discussed for SSAs in section 9.2.4 (p. 191), the inclusion of $\bar{e}$ may still yield valuable insights: if it is the prevailing factor in determining the most suitable algorithm, this motivates the development of synchronization

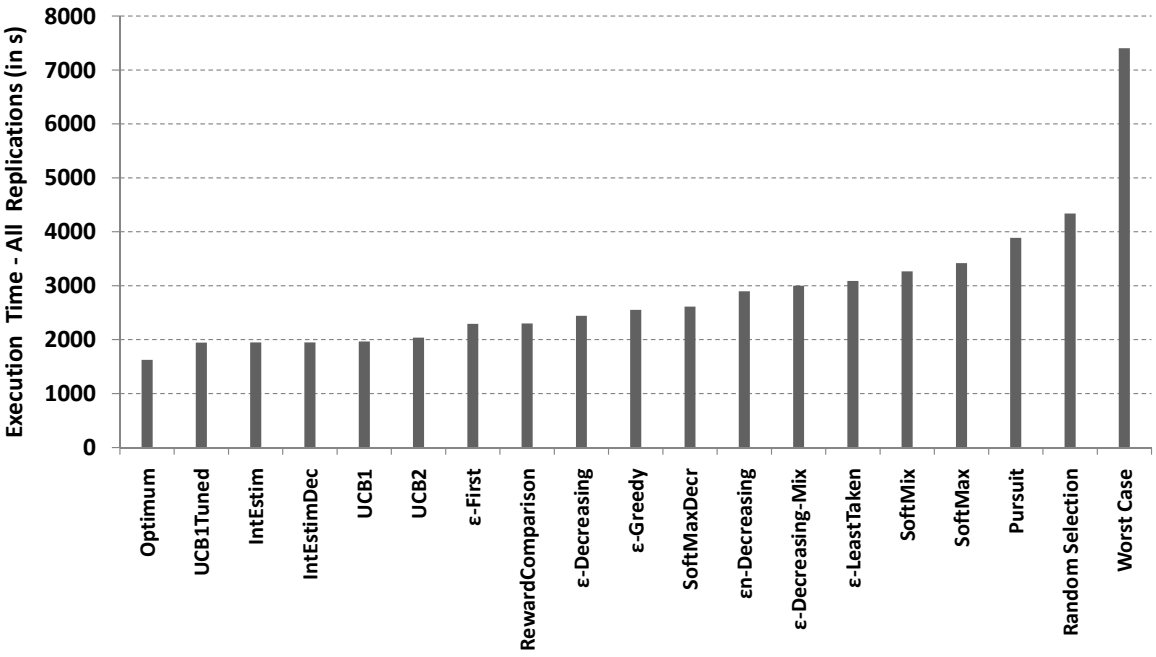

Figure 10.6.: Overall execution time of policies on PHOLD benchmark experiment: All policies outperform RandomSelection by at least 11%. Worst-case and optimum performance have been estimated again (see sec. 9.2.3, p. 187).

protocols that adapt to $\bar{e}$ at runtime. Recall that $\bar{e}$ is directly specified as a PHOLD parameter (see sec. 10.2.1, p. 200); for the performance exploration it has been chosen from $\{\frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$ (sec. 10.2.2, p. 201).

Similar to section 9.2.4 (p. 191), the performance of the generated selectors is at first estimated via 10 replications of bootstrapping (see sec. 6.2.2, p. 124), applied to the performance data from the experiment described in section 10.2.2 (p. 201). Figure 10.7 (p. 206) shows the results. Similar to the results presented in section 9.2.4 (p. 192), Naïve Bayes does not even outperform a random selection for boolean mispredictions (upper left plot, fig. 10.7), which is again due to the particular problem structure. The best methods — M5P and J48 — select the most suitable algorithm in over 70% of all trials, more often than the winner-takes-all (WTA) approach. Still, none of the selectors that rely on quantitative performance prediction is able to achieve an acceptable accuracy; their average prediction error varies between 66% and 69% of the actual run time (upper right plot, fig. 10.7). The misprediction regret, which estimates the performance loss due to mispredictions, show that the use of tree-based selectors – M5P, J48, and ID3 – seems most promising besides the use of the WTA selector. The strong error of neural network selectors, which perform worse than a random selection, suggests that there are too few performance data to properly train a single network for decision-making.

The effectiveness of the generated selectors in practice is evaluated on a sample experiment, which comprises eight PHOLD setups that have not been considered by the selectors so far. The eight setups are defined by choosing the number of LPs (n_{LP}) from $\{225, 400\}$, the average initial number of events per LP ($\bar{e}$) from $\{0.6, 1.2\}$, and setting the topology to either `FULL` or `GRID`. Hence, some of the unseen PHOLD setups are a little larger than those in the training set (400 instead of 256 LPs) and also exhibit a little more inherent parallelism and load ($\bar{e} = 1.2$ instead of 1).

Figure 10.8 (p. 207) shows the results for ten replications of each experiment. An experiment starts out with the generation of a new selector that is now trained with *all* available performance data from section 10.2.2 (p. 201). The `EnsembleSelectorGenerator` relying upon neural networks has a

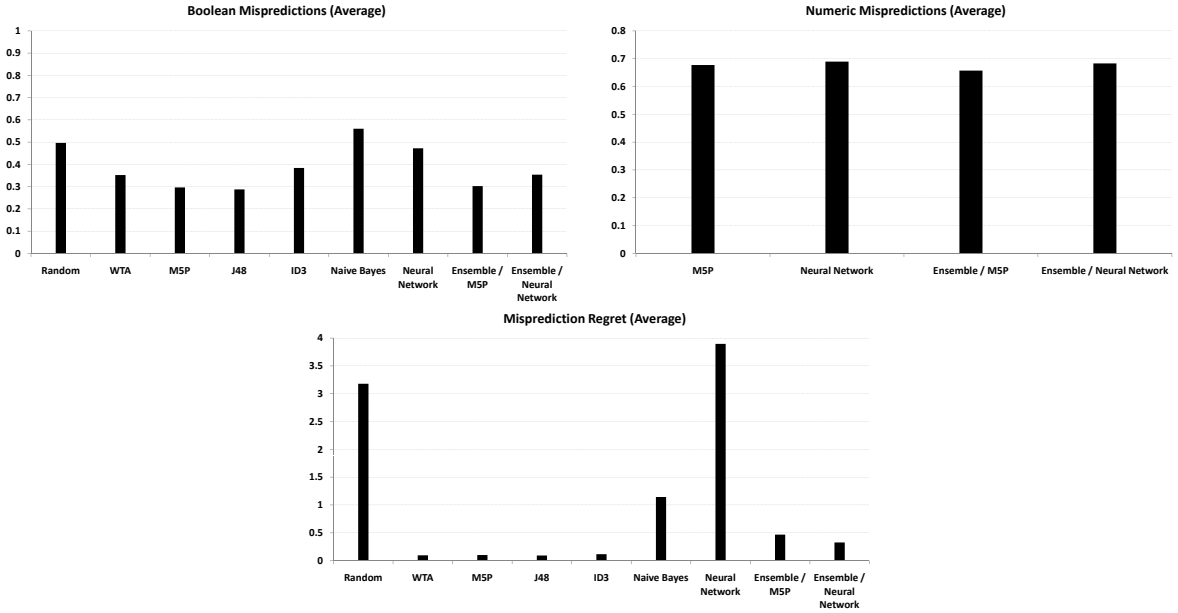

Figure 10.7.: Selector performance estimated by the measures from section 6.2.2 (p. 125), similar to figure 9.9 (p. 192). Again, the numeric misprediction measure only applies to methods that give quantitative predictions. Misprediction regret (bottom) plots the performance overhead with respect to an optimal selection (i.e., S^*) in percent.

huge standard deviation: sometimes its choice is almost optimal, but sometimes it is very bad. This behavior is due to the random initialization of initial node weights that is carried out by JOONE's back-propagation algorithm to train the neural networks. Sometimes, this leads to a neural network for the null message protocol that overestimates the performance on setups where NM is clearly inferior; for one setup it is almost *fifty* times slower than the best choice! These large performance differences between the algorithms also explain why the performance deviation of randomly selecting an algorithm is rather large as well.

Unlike the results shown for SSAs (see fig. 9.10, p. 194), all other selectors manage to outperform the WTA approach in this setting. In fact, some even are adaptive-effective for the considered simulation problems (see def. 2.1.6, p. 18): while the BTB configuration selected by the WTA is not the best-performing algorithm for the eight setups in question, even the best constant selector (which would always choose barrier synchronization) is $\approx 11\%$ *slower* than the ID3 selector, which performs best here. In other words, no fixed selection mapping is able to outperform the best-performing selector that takes into account some model features for selection. In fact, both ID3 and J48 select the *optimal* algorithm for all eight setups; M5P gets it right six out of eight times. ID3 speeds up the experiment by a factor of ≈ 1.8 (w.r.t. WTA) and ≈ 4 (w.r.t. random selection). All these results show that an automatic algorithm selection *can* be very efficient in practice and *may* speed up a simulation system considerably, as long as suitable features are available and no algorithm is too dominant (as in case of the SSAs, sec. 9.2.4, p. 191). However, it should be kept in mind that this is only a *very* small case study, operating on a small set of algorithms and only concerned with relatively few setups of a single benchmark model.

10.3. Summary

Similar to chapter 9 (p. 177), this chapter illustrates the benefits and shortcomings of the techniques presented in chapter 5 (p. 101) to chapter 8 (p. 159), now restricted to the main methodological

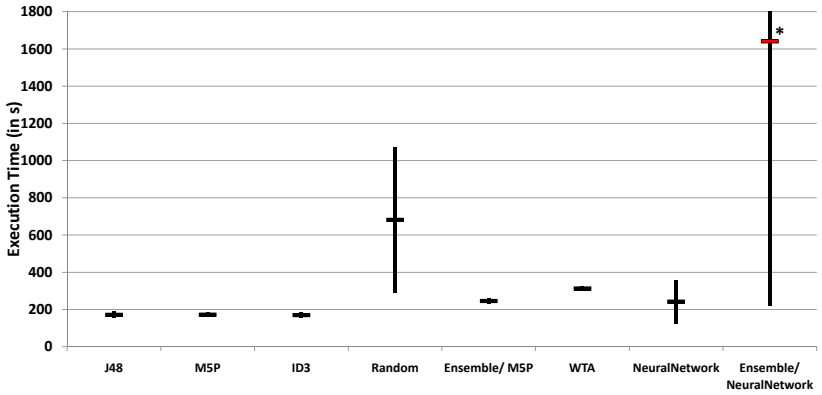

Figure 10.8.: Real-world performance of selectors generated by different mechanisms, executed on eight formerly unseen PHOLD setups. Each experiment has been replicated ten times. Horizontal bars denote the average execution time, vertical bars display the standard deviation in both directions (i.e., average execution time $\pm\sigma$). The only exception to this is the rightmost data point, marked red and with an asterisk (`EnsembleSelectorGenerator` with neural networks), where the standard deviation is so high that the average execution time minus the standard deviation is *negative*. Here, the vertical line goes down to the minimal run time that was achieved; the maximum run time lies at 3750.7 s (not displayed).

challenges: the collection off sufficient amounts of performance data (sec. 10.2.2, p. 201), algorithm selection without any prior knowledge (sec. 10.2.3, p. 202), and finally algorithm selection with prior knowledge (sec. 10.2.4, p. 203), which applies machine learning to generate suitable algorithm selectors.

While all other techniques have already been successfully applied in chapter 9 (p. 177), the last section could also show that such learned selectors are indeed able to improve the overall performance of a simulation system – even when compared with the best possible constant selection mapping (see sec. 2.1.2, p. 16), which is usually unknown. While the general effectiveness of this procedure is already shown in section 8.2.2 (p. 169), section 10.2.4 (p. 203) illustrates that performance gains due to sophisticated algorithm selection are also attainable in practice.

Nevertheless, this whole study should be considered as a proof of concept: as with the SSA case study (ch. 9, p. 177), the presented results are hard to generalize and only regard a fairly small algorithm selection problem, i.e., few algorithms, few features, and few model setups. And even here, further methodological problems can be noticed: the measures used to evaluate the generated selectors (see fig. 10.7, p. 206) do *not* allow to anticipate the bad performance of the `EnsembleSelectorGenerator` with neural networks. One way to alleviate such problems when deploying selectors to users would be to employ meta-learning (see fig. 2.15, p. 59). These and some other directions of future work will be discussed in the following chapter, which concludes the thesis.

11. Conclusions

> *The truth will set you free. But not until it is finished with you.*
> David Foster Wallace, *Infinite Jest*, 1996

As motivated in chapter 1 (p. 1), this thesis covers mechanisms to automatically select suitable simulation algorithms for complex simulation problems, i.e., problems for which simulator performance is hard to foresee. This chapter concludes the thesis. Section 11.1 summarizes the main methodological contributions, introduced in chapter 4 (p. 79) to chapter 8 (p. 159). Besides relating them to the concepts presented in chapter 2 (p. 13) and chapter 3 (p. 61), their relative merits and drawbacks are discussed by analyzing the results from the case studies (ch. 9, p. 177, and ch. 10, p. 197). A more succinct list of theses can be found in section A.1 (p. 219). Open questions and future research directions are detailed in section 11.2.

11.1. Summary

The main contribution of this thesis are two distinct approaches for simulation algorithm selection. Their principal difference is in the reliance on prior efforts: with a sufficient amount of past performance data and suitable problem features, it is possible to construct selection mappings that select a suitable simulation algorithm *before* execution. If this data is not available, a suitable algorithm can still be selected in case of stochastic simulation, i.e., in the presence of multiple replications. The latter approach can be improved by taking into account previous performance data *without* problem features. Figure 11.1 outlines the three situations that can be distinguished overall: having no data (using adaptive replication), having past performance data (using adaptive replication and portfolio selection), and having both past performance data and suitable problem features (using generated selectors).

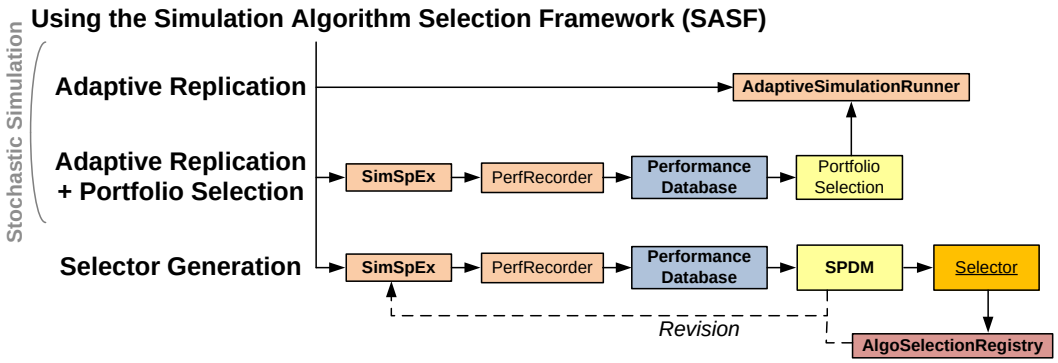

Figure 11.1.: Three ways of solving the algorithm selection problem with the SASF components from figure 8.10 (p. 172). If no prior performance data is available, it is still possible to use the adaptive simulation runner (provided a sufficient number of replications). Portfolio selection may improve the convergence of the adaptive simulation runner, but requires performance data. Finally, in case there are also feature extractors defined for the simulation problems, the framework for simulator performance data mining (SPDM) can be used to construct a suitable selection mapping, i.e., a `Selector`. A selector should be assessed on *unseen* problems, in order to decide whether its performance is satisfactory or a revision is necessary (e.g., considering new features or simulation problems).

All components operate on the same basic entities — e.g., runtime configurations (sec. 5.1.1, p. 104) — and are organized within the simulation algorithm selection framework (SASF, ch. 4, p. 79). Adaptive replication does not require to involve simulation experts (i.e., developers, performance analysts, or deployers — see sec. 4.1, p. 79). It is relatively straightforward to set up and hence requires little effort. The downsides are its restriction to multiple simulation replications, i.e., stochastic simulation, as well as the irreducible overhead due to exploration (see sec. 2.3.2, p. 29). The latter can be reduced by constructing portfolios of simulation algorithms (sec. 7.2, p. 134), as shown in section 9.2.3 (p. 188).

The alternative is selector generation, i.e., to conduct a full-fledged performance study that includes the identification and automatic extraction of important problem features. The collected data can then be analyzed by the SPDM (ch. 6, p. 115) and deployed to the `AlgoSelectionRegistry` (ch. 8, p. 159). While this method is generally applicable and predominant in the literature (see ch. 2, p. 13), it also requires additional efforts.

Categorization: Putting the Developed Methods into a General Context

Figure 11.2 (p. 211) shows how the algorithm selection approaches provided by the SASF can be categorized, regarding the aspects that have been described in section 2.6 (p. 47). The reasoning behind the categorization is as follows:

- **Problem**
  The SASF allows to implement approaches for *optimization*-type selection problems, i.e., instead of asking whether an algorithm may reach a certain performance level, the *best-performing* algorithm shall be selected. Adaptive replication tries to find the *single* most suitable simulation algorithm by considering a *set* of simulation problem instances (see discussion in sec. 2.6.1, p. 48). The selectors generated by the SPDM choose a *single* algorithm by considering the features of a *single* simulation problem instance. The mechanism for portfolio selection supports the selection of an algorithm *set* to cope with a *set* of future problems (see fig. 11.2, p. 211).

- **Algorithms:** As figure 11.2 also shows, the focus of the SASF is on *combinatorial-parametric* algorithms. This suits the abstraction level provided by JAMES II plug-ins and plug-in types. Depending on the implementation of a simulation algorithm in JAMES II, it may also be regarded as *monolithic* or *monolithic-parametric*. However, JAMES II does not support the automated *generation* of algorithms so far, hence no specific techniques have been developed in that regard. Moreover, no custom mechanism to explore the parameter space of a single algorithm is available yet.

- **Data:** The SASF is focused on the analysis of *empirical* data (see sec. 1.4, p. 8, and sec. 2.8, p. 58); it does not support the inclusion of *theoretical* findings as such. Nevertheless, theoretical knowledge can be leveraged by defining suitable problem features, by employing customized methods for data analysis, or by implementing specific selectors that exploit such knowledge. The SASF does not focus on any particular type of performance data, i.e., it may come from *realistic* or *synthetic* problems. Yet, the exploration of a simulation space large enough to warrant data analysis with the SPDM is likely to require a focus on *synthetic* problems that are representative to some extent (see discussion in sec. 7.3.1, p. 145). The experiments presented in section 8.2 (p. 166), chapter 9 (p. 177), and chapter 10 (p. 197) all relied on *synthetic* benchmark models and considered platform-dependent performance. Algorithm performance is likely to vary across different hardware *platforms* (e.g., due to differing CPU product lines), but should be less dependent on more specific machine properties, such as the length of the GPU's graphics pipeline (unless it is used for execution). While the adaptive simulation runner collects data *continuously* and automatically, i.e., during the execution, the usual way of collecting performance data with the SASF is *triggered* by the user: the performance recorder (sec. 5.2, p. 112) needs to be activated.

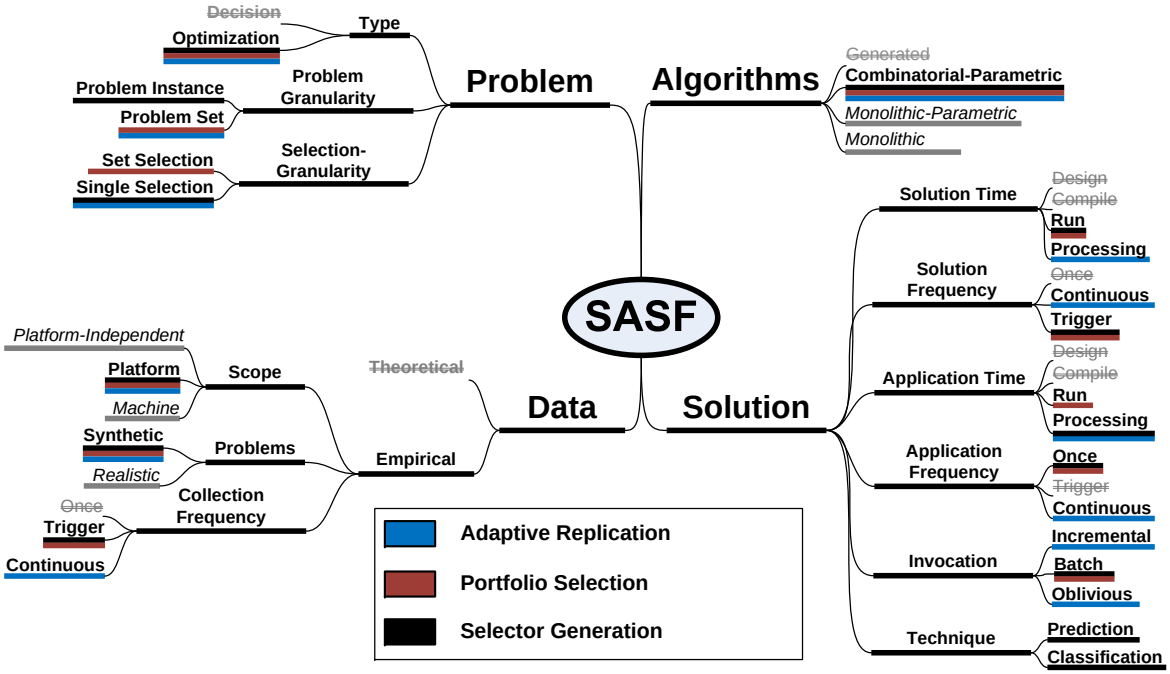

Figure 11.2.: Categorization of the methods provided by the simulation algorithm selection framework, following figure 2.13 (p. 52) and the categorization in section 2.6 (p. 47). The SASF does not offer support for the categories that are crossed out. Categories that are underlined in gray apply to all approaches, but are not explicitly addressed by the SASF. The solution technique categories (prediction and classification, see sec. 2.6.1, p. 50, and fig. 6.2, p. 116) are not meaningful for adaptive replication and portfolio selection. During adaptive replication, the adaptive simulation runner (sec. 7.2, p. 134) learns incrementally which algorithm works best for the given simulation problem, but this knowledge is dismissed when moving to the next problem — hence the approach is both incremental *and* oblivious (in terms of solution invocation, also see sec. 2.6.1, p. 50).

- **Solution:** The solution approach of the adaptive simulation runner can be regarded as searching for the best constant selection mapping while the simulation problem is being processed (see discussion in sec. 2.3.2, p. 32). It hence *continuously* solves the algorithm selection problem and applies its current solution, both at *processing time*.[1] It also works *obliviously*, i.e., selection mappings are dismissed whenever a new simulation problem is encountered. In contrast, portfolio selection is typically conducted at *run time*; it does not consider any features of the given simulation problem. Similarly, SPDM's selector generation is *triggered* by a user — more specifically, a deployer — at *run time*. The resulting selector is applied only *once* per simulation problem. So far, no incremental learning algorithms have been considered, so the SPDM solves the algorithm selection problem by analyzing a *batch* of performance data. The SASF supports both *performance prediction* and *problem classification* (see sec. 6.2.1, p. 119). This is achieved in the SPDM by encapsulating the selection logic within a simple Java `Comparator` (see sec. 6.2.1, p. 119, and fig. 6.6, p. 120).

[1] Note that this refers to the processing time of a simulation problem, *not* a simulation problem instance (see sec. 2.4.1, p. 34, and sec. 2.6.1, p. 50). Solving the ASP at the processing time of a simulation problem *instance* can be done as well; it implies to switch algorithms during the execution of a single simulation run.

Experiment Results

The general effectiveness of the SASF regarding its ability to automate performance data collection (ch. 5, p. 101, and ch. 7, p. 131), data analysis (ch. 6, p. 115), and finally simulation algorithm selection (sec. 8.1, p. 159) has been illustrated in section 8.2 (p. 166). Here, the major result is that — given appropriate problem features and suitable SPDM selector generators — the performance of the `AlgoSelectionRegistry` is rather close to optimal, with overhead being less than 13% away from optimum (on average). However, such a synthetic setup does not illustrate how well-suited the developed methods are for real-world problems; this has been investigated in chapter 9 (p. 177) and chapter 10 (p. 197).

Section 9.2.2 (p. 181) highlights the importance of advanced techniques for simulator performance evaluation: naïve setups either slowed down experiment execution by almost one order of magnitude (fig. 9.2, p. 183) or resulted in noisy data (fig. 9.4, p. 185). In chapter 10, the same experimentation techniques — the adaptive simulation runner (sec. 7.2, p. 134) combined with a suitable benchmark model and simulation end time calibration (sec. 7.3, p. 145) — are also applied successfully (see sec. 10.2.2, p. 201).

The adaptive simulation runner is not only useful for performance experiments, but also for algorithm selection without prior knowledge. The effectiveness of such an adaptive replication (see fig. 11.1, p. 209) is shown both in section 9.2.3 (p. 186) and section 10.2.3 (p. 202). A speed-up of 1.77 and 2.2 could be achieved, respectively (compared to RandomSelection). Note that speed-up by algorithm selection is gained *without* adapting the algorithms or the hardware; it solely stems from the flexibility of the simulation system.

While the performance of most policies for adaptive replication varies across both case studies, some policies like IntEstim were found to deliver good performance in both cases. However, this does not ensure their suitability for other simulation studies. On the other hand, *all* policies outperformed a random algorithm selection over the course of a whole experiment — so there seems to be only little risk attached with adaptive replication. Section 9.2.3 also shows that prior portfolio selection (as described in sec. 7.2.2, p. 140) may even result in near-optimal performance (fig. 9.8, p. 191).

Finally, the effectiveness of the SPDM and the `AlgoSelectionRegistry` is evaluated in section 9.2.4 (p. 191) and section 10.2.4 (p. 203). In contrast to the synthetic setup considered in section 8.2 (p. 166), the results on real-world problems are mixed; some of the selectors performed rather bad — sometimes *worse* than random choice (e.g., see fig. 10.8, p. 207). Still, most selectors outperformed a random algorithm selection. Apart from the simple winner-takes-all approach, the tree-based methods (M5P, J48, and ID3) exhibited the best real-world performance in both case studies.

In the first case study, one algorithm (ODM, sec. 9.1, p. 179) almost dominated the problem space under scrutiny. None of the other SSA algorithms performed really bad; one reason for this is the restriction to those JAMES II event queues that performed well in a previous study (see discussion in sec. 9.1, p. 179). These circumstances make simulation algorithm selection quite challenging and prevented the generated selectors from being adaptive-effective, i.e., to outperform the best-performing constant selection mapping (see def. 2.1.6, p. 18). As such a mapping is typically *unknown* a priori, this does *not* mean the SPDM has failed — it rather shows that the *given* machine learning methods cannot derive better selection mappings from the *given* performance tuples (see def. 2.3.1, p. 24). This either motivates the consideration of additional features and problems (see fig. 11.1, p. 209) to generate better selectors, or the deployment of the winner-takes-all strategy (because it works well enough). End users can benefit from the deployed selector, no matter if it relies on problem features or not — and even here, the best selector could achieve a speed-up of ≈ 1.5 (w.r.t. random choice).

In the second case study (sec. 10.2.4, p. 203), two selectors are adaptive-effective for the given sample experiment, i.e., they outperform *any* fixed selection mapping because they consider problem features. The selectors achieved a speed-up of ≈ 4 (w.r.t. random choice), and could be shown to make *optimal* choices during evaluation.

While the observed speed-ups for SPDM selectors should be interpreted carefully — there was only one benchmark model per case study and the test experiments were rather small — they still

hint at the potential performance benefits of simulation algorithm selection. This is because there are many aspects that are likely to make the SPDM work *better* when applied at a larger scale in practice. Firstly, both case studies collected relatively little performance data, and many potential problem features have been excluded from analysis. Secondly, there are many other machine learning approaches that may generate better selectors, and the ones that were used have not been fine-tuned during the experiments. This leaves much room for improvement. Finally, the performance differences of the investigated algorithms were rather small — ongoing work in the domain of spatial SSA variants is confronted with problems where one algorithm may be several *thousand* times slower than another. It is a promising future application domain for simulation algorithm selection, as preliminary results from performance data mining suggest that the construction of suitable selection mappings is feasible.

General Ramifications

Now that it is clear what the mechanisms developed in chapters 4 to 8 mainly provide, how they can be realized, and how they perform, the question remains which *general* conclusions can be drawn for the field of simulation. The flexibility of JAMES II and the way plug-ins are combined make algorithm selection particularly challenging (see sec. 4.2.3, p. 87, and sec. 4.3, p. 89). As the presented mechanisms to support simulation algorithm selection were developed for a host system of this complexity, other simulation systems of similar (or less) complexity may rely on the same (or simplified) techniques. This should go without additional work on the principal structure of the SASF.

An interesting aspect of the SASF is the variety of elements it relies upon. It is already proposed by Rice [272] that algorithm selection requires the collection of data (ch. 5, p. 101) and their analysis to construct suitable selection mappings (ch. 6, p. 115). It becomes apparent that, in order to successfully realize simulation algorithm selection *in practice*, two additional aspects need to be considered:

- **Experimental algorithmics**: The analysis of simulation algorithm performance is itself complicated and time-consuming (see ch. 3, p. 61). Without efficient techniques for experimentation, the collection of sufficient performance data would be prohibitively time-consuming and cumbersome. Additional experimentation mechanisms, motivated by simulation algorithm selection, are hence covered in chapter 7 (p. 131).

- **Deployment**: It is *not* sufficient to just generate a selection mapping, it also needs to be deployed in a form that suits the host system. Otherwise, the knowledge gained by analyzing the past performance data cannot be exploited in practice. Several requirements arise in this context, e.g., it may be necessary to manage multiple selection mappings,[2] or to ensure a seamless integration of the selection mechanism. The latter is particularly important if simulation algorithm selection shall be realized for an *existing* system, so that massive changes to the code base (and documentation, etc.) are avoided. Finally, non-quantifiable performance measures (see sec. 5.1.1, p. 108) may need to be considered as well, e.g., to prevent the selection of an algorithm that is unstable.

  All these issues have been addressed by a prototypical extension of the JAMES II registry, which can manage multiple selectors, makes algorithm selection transparent to other JAMES II code, and keeps track of plug-in status (ch. 8, p. 159). The status of a plug-in can even be reset automatically by a simple failure detection mechanism (sec. 8.1.2, p. 162).

The implementation of all these elements requires some effort. On the other hand, it becomes clear from the use case analysis in section 4.1 (p. 79) that not only end users — i.e., users in the role of experimenters — benefit from the developed components: both simulation algorithm developers and performance analysts may also employ them for performance data management and analysis. This enables a stronger focus on algorithms of practical relevance and may make performance analyses faster, easier to conduct, and more powerful. One example for this is the simple component to

[2] For example, to cover different subspaces of the problem space $\mathbb{P}$; e.g., one selector for SSAs (sec. 9.2.4, p. 191) and another one for PDES (sec. 10.2.4, p. 203).

estimate the effects of code changes on the overall performance of the simulation system (sec. 7.3.4, p. 156). All this is made possible by treating simulation algorithms as what they are: not just some code fragments, but independent abstract entities that deserve careful evaluation and exploration.

Finally, providing means for automated simulation algorithm selection seems to be a fundamental requirement for any simulation system that is sufficiently flexible. Flexibility and extensibility foster software re-use and should hence be advocated in any case. As this thesis shows how to complement flexible simulation systems with mechanisms for simulation algorithm selection, it also demonstrates how to improve the overall performance of such simulation systems *beyond* the performance of systems that only provide single monolithic implementations. In this sense, the results of this thesis accentuate the importance of flexibility in simulation systems, while alleviating one of their major weaknesses: increased complexity of configuration. What is the point of having a flexible simulation system if no end user is able to exploit it? Since the presented techniques help to overcome this problem, they may also help the proliferation of such simulation systems.

Limitations

As discussed before, this thesis contains some results with interesting general implications, e.g., regarding the development practices for simulation algorithms or the merits of flexible simulation systems. However, it is equally important to see the *limitations* of these findings.

First of all it should be noted that the SASF, although being tested and developed for many months, still is a research *prototype*, in the same sense that its host system JAMES II still is a research prototype: interfaces may turn out to be incomplete or ill-suited, components might need to be added, others might need refinement or refactoring. This is in itself not unusual — as DeMarco puts it: *"Software development is and always will be somewhat experimental. The actual software construction isn't necessarily experimental, but its conception is. And this is where our focus ought to be"* [57, p. 95]. At least there is reasonable hope that all *essential* entities — e.g., feature extractors, selector generators, selection trees — have been identified, since the system has been shown to work effectively in principle (sec. 8.2, p. 166).

However, neither the impact of hardware changes nor the handling of multiple user criteria has been evaluated so far. The impact of hardware is unlikely to yield much additional insights regarding the effectiveness of the developed methods, since the data is already regarded as platform-dependent (see fig. 11.2, p. 211). The developed methods can cope with multiple user criteria in principle, but this makes their evaluation much more challenging (see discussion on SSA accuracy in sec. 9, p. 177). Therefore, this aspect has been left out so far and needs to be addressed in the future (see sec. 11.2, p. 215).

Another aspect to be examined critically is the *automation* of algorithm selection: as described in section 4.2 (p. 81), JAMES II already has an automatic (yet hard-coded) algorithm selection mechanism in place. One could even argue that some efforts required for the newly developed methods, e.g., the configuration of suitable performance experiments, are only carried out semi-automatically so far — hence the approaches presented here make algorithm selection *less* automated than it was before! While it is certainly true that manual intervention is *still* required for specific tasks (another example would be the deployment of selectors to the `AlgoSelectionRegistry`), these steps have to be carried out by the *deployers* of JAMES II, not by its end users (i.e., experimenters). Further automation and simplification for all user roles seems likely and straightforward, e.g., by integrating more powerful techniques for experiment design (see sec. 3.2, p. 66). Moreover, automatic algorithm selection is only beneficial if *suitable* algorithms are selected. So far, JAMES II only supported a *fixed* selection from all algorithms that are eligible to solve a problem. For example, event queue X was *always* preferred over event queue Y, regardless of the model to be simulated or the simulator to be used (see sec. 4.2.3, p. 87).

Finally, it has to be conceded that the settings in which algorithm selection was shown to work are still rather small, both in terms of observed algorithms and performance data. This is particularly true for the second case study on parallel discrete-event simulators. Parts of the SASF have also been

applied to other problem domains, e.g., cellular automata [280] or spatial SSA extensions, and thereby already facilitated research on simulation algorithms [77]. More elaborate studies are required to give a better impression of the *overall* SASF effectiveness in various application domains. All machinery is in place to do so in the future.

11.2. Outlook

Due to the breadth of the algorithm selection problem, many questions have to be left unanswered for now. Besides research directions that were explicitly left out of the discussion — e.g., solutions for the best features for algorithms problem (def. 2.1.3, p. 15) — several other dimensions of simulation algorithm selection have not been covered in the thesis. They can be easily recognized by considering the categorization of the current mechanisms, as given by figure 11.2 (p. 211):

- **Data**: The inclusion of analytical findings has not been considered yet, although there are some theoretical results that may help designing performance experiments, or could even complement selection mappings derived from empirical data. Examples of interesting approaches to analytical simulation algorithm analysis have been briefly discussed in section 3.3.1 (p. 71).
- **Algorithms**: While algorithms in JAMES II are combinatorial-parametric (sec. 2.6.1, p. 49), i.e., JAMES II plug-ins have parameters and can be combined to some extent, neither the structure of the resulting selection trees (see def. 5.1.1, p. 105) nor algorithm parameters have been taken into account for specific algorithm selection techniques. Such advanced techniques could, for example, exploit additional knowledge on parameters (e.g., performance sensitivity) to further improve algorithm selection.
- **Solution**: Incremental learning is supported by the adaptive simulation runner (see sec. 7.2, p. 134), but only in an oblivious manner (i.e., the results are dismissed after the simulation problem has been processed, see sec. 2.6.1, p. 50). Fully automated algorithm selection would have to provide non-oblivious incremental learning that is carried out in the background. This can be done by including meta-learning, e.g., in the way it has been used to extend the AOTA framework (see fig. 2.10, p. 44). Furthermore, if both solving the ASP and applying the solution shall be done continuously at processing time, this is currently only supported by the adaptive simulation runner and hence only applicable for stochastic simulations. Of course, it could also be beneficial to switch the simulation algorithm *during* execution. While approaches that adjust simulation algorithms during execution have already been developed (e.g., [323, 231]), these are tailored towards specific algorithms and do not consider prior performance data. Another important aspect is the algorithm selection based on *multiple* user criteria — while the developed methods are (mostly) able to cope with this requirement, they may still need some enhancements in this regard. Finally, it should be investigated to what extent meta-learning and empirical tuning may further improve the performance of a simulation system, by providing complementary algorithm selection solutions on adjacent levels (as shown in fig. 2.15, p. 59).

Apart from these rather broad issues, some more specific questions arose in the chapters presenting the SASF components in detail (i.e., ch. 5, p. 101, to ch. 8, p. 159). These are listed in the following:

Storage of Performance Data (ch. 5, p. 101)

While the performance database as such is working well, there are many additional features that would make its usage much easier. First of all, a dedicated user interface would greatly facilitate the management of performance data. Together with an automated plotting or reporting mechanism, it could be used to efficiently share results among peers. Additional visualization techniques can help to grasp the performance trade-offs between different algorithms. Another important step would be to make central instances of the performance database accessible over the Internet, in order to

build a common repository for performance results. Besides allowing users to incorporate platform-independent performance data that was generated by others, this would also make simulation research more transparent (by providing a general and quantitative overview of different methods).

Selection Mapping Generation (ch. 6, p. 115)

There are many mechanisms to generate selectors — only few of them have been applied in this thesis. Future work should strive to explore other mechanisms, so that the most suitable ones can be identified. This also implies the development of additional performance metrics (besides those defined in sec. 6.2.2, p. 125) to estimate the performance of the generated selectors beforehand. The SPDM also requires a graphical user interface, which should be integrated with that of the performance database (e.g., in the spirit of PerfExplorer [150], see sec. 3.3.2, p. 73). Moreover, the feature selection problem (see def. 2.1.3, p. 15) should be addressed explicitly.

Experimentation Methodology (ch. 7, p. 131)

The different mechanisms presented in chapter 7 (p. 131) leave (arguably) the most room for further improvement and investigation. In case of the adaptive simulation runner, further research should be focused on the pre-selection of simulation algorithm portfolios. Alternative portfolio approaches should be compared with the existing ones; some of them should be able to consider multiple user criteria (as discussed in sec. 2.5.4, p. 46, and sec. 7.2.2, p. 141). Another interesting problem is to automatically determine the *size* of a portfolio, given the simulation problem to be solved and the number of replications that shall be executed. Further research may also investigate ways to incrementally select portfolios, i.e., to make performance recording and subsequent portfolio construction transparent to the user. This issue is closely related to the question of how much data will be necessary to select portfolios of a certain quality.

The components for simulation space exploration (sec. 7.3.3, p. 153) allow to incorporate more advanced techniques for experiment design and meta-modeling, in order to further automate simulation algorithm performance analysis. For example, a novel algorithm to find performance trade-off points for a set of simulation algorithms is already under development. Other work could focus on exploring the parameter space of an algorithm, which may yield additional feedback for developers. Finally, the algorithmic change evaluator (sec. 7.3.4, p. 156) — the only component that specifically addresses the needs of developers so far — should be easier to use: it needs a graphical user interface, preferably one that can be accessed easily from an integrated development environment.

Automatic Simulation Algorithm Selection in JAMES II (ch. 8, p. 159)

The `AlgoSelectionRegistry`, as described in chapter 8 (p. 159), is just a research prototype. Here, some more practical programming issues should be solved in order to establish algorithm selection as a basic service within the JAMES II core. More advanced schemes should be developed to handle failure detection, and the status of all plug-ins should be visible to the user, i.e., the plug-in data storage should be maintainable through the general JAMES II user interface.

Looking Ahead...

All of the above issues present challenging future work, which cannot (and should not) be carried out by a single researcher or research group. Instead, algorithm selection should be regarded as a major problem to be dealt with in *all* simulation systems with a certain flexibility. If anything, this thesis shows the potential benefits of explicitly addressing the problem from a simulation viewpoint.

Ultimately, simulation research shall improve and enhance the usage of simulation *in practice* (see sec. 1.1). Automatic simulation algorithm selection helps to do so — it should be worth the trouble!

Acknowledgements

Many people have supported me during the last four years. I want to start with thanking my supervisor Lin Uhrmacher for her enduring encouragement and guidance. The same goes for all current and former friends and colleagues from the modeling and simulation group at the University of Rostock; it was great fun to work with you and I learned a lot! You know how dubious I find most rankings — and how to value your immeasurable support? — so here you are in alphabetical order: Alexander Steiniger, Alke Martens, Anja Hampel, Arne Bittig, Carsten Maus, Fiete Haack, Florian Marquardt, Jan Himmelspach, Mathias John, Mathias Röhl, Matthias Jeschke, Nadja Schlungbaum, Orianne Mazemondet, Sigrun Hoffmann, Stefan Leye, Stefan Rybacki, and Susanne Jürgensmann. And let's not forget Fritz ;-)

Special thanks go to Stefan Leye, who took over teaching one of my exercise groups during the last semester of writing this, to Matthias Jeschke, who provided many interesting SSA algorithms for evaluation, and to my officemate Jan Himmelspach. I would also like to thank Bing Wang, who developed the PDES algorithms I evaluated in chapter 10 (p. 197). Kaustav Saha and Steffen Torbahn helped me with implementing some parts of the SPDM; René Schulz implemented some of the multi-armed bandit policies and worked with me on the genetic algorithm for portfolio selection (sec. 7.2, p. 134).

While succeeding at work is one thing, staying sane while going through this is another. I am deeply grateful to my family; for their support, their patience, and their understanding.

A. Appendix

A.1. Theses

1. Simulation algorithm selection is necessary to support end users in their application of simulation tools. It is particularly important in case many different algorithms are available and the relative performance benefits of the simulation algorithms are not fully understood.

2. The algorithm selection problem has been addressed in many fields, which use distinct terminology for similar concepts. A categorization for algorithm selection approaches facilitates their assessment and comparison.

3. Simulation algorithm selection may be supported by techniques for performance prediction and experiment design.

4. Selection mappings for simulation algorithms can be generated by analyzing past performance data, e.g., with methods from machine learning. To do so, the performance data needs to be associated with suitable problem features, so that a relation between both can be established empirically.

5. The generation of suitable selection mappings requires considerable amounts of performance data, which should be stored in a dedicated data sink.

6. Experimental setups tailored towards algorithm selection allow an efficient collection of the required data. This includes the usage of suitable synthetic benchmark models. The models need to exhibit a quasi-steady state and should be scalable, parameterizable, and simple to implement.

7. Adaptive simulation replication is able to speed up stochastic simulation experiments without relying on past performance data.

8. Portfolios of simulation algorithms can be pre-selected, based on past performance data. This can increase the convergence speed of adaptive simulation replication.

9. The methods implemented within the simulation algorithm selection framework (SASF) allow to improve the overall performance of simulation systems, as long as these are flexible and their application domain is not dominated by a single algorithm.

10. Methods for simulation algorithm selection are able to support the development and evaluation of new simulation methods. This efficiency gain in investigating simulation methods can also be beneficial for end users, as it may lead to improved implementations.

A.2. Proof: Average and Adaptive Effectiveness

Theorem A.2.1. *If a selection mapping S' is adaptive-effective (def. 2.1.6, p. 18) for a certain problem set $P \subseteq \mathbb{P}$ and algorithm set $A \subseteq \mathbb{A}$, it is also average-effective (def. 2.1.5, p. 17) for P and A.*

Proof. It has to be shown that

$$\overline{perf}(S', P) > \frac{\sum_{a \in A} \overline{perf}(S_a, P)}{|A|} \tag{A.1}$$

holds.

Let $\hat{p} = \max_{a \in A}\{\overline{perf}(S_a, P)\}$ be the performance of the best constant selection mapping S^*_C (see def. 2.1.8, p. 19), i.e., $\hat{p} = \max_{S \in \mathbb{S}_C}\{\overline{perf}(S, P)\} = \overline{perf}(S^*_C, P)$. Since $\hat{p} \geq \overline{perf}(S_a, P)$ holds for all $a \in A$, an upper bound for the right-hand side of equation A.1 can be written as:

$$\frac{\sum_{a \in A} \overline{perf}(S_a, P)}{|A|} \leq \frac{|A| \cdot \hat{p}}{|A|} = \hat{p} \tag{A.2}$$

By definition 2.1.6 (p. 18), the following holds for S':

$$\forall S \in \mathbb{S}_C : \overline{perf}(S', P) > \overline{perf}(S, P) \tag{A.3}$$

Equation A.3 implies that $\overline{perf}(S', P) > \overline{perf}(S^*_C, P) = \hat{p}$ (see above), so that a combination with A.2 yields:

$$\overline{perf}(S', P) > \overline{perf}(S^*_C, P) = \hat{p} \geq \frac{\sum_{a \in A} \overline{perf}(S_a, P)}{|A|}$$

□

A.3. Categorization of Algorithm Selection Approaches

Presented in	Domain	Problem: Type	Problem: Granularity	Problem: Selection	Data: Dependence	Data: Frequency	Algorithms	Solution: Time	Solution: Frequency
[25]	Sorting	Opt.	Instance	Single	Machine	Once	Mon/Params	Compile	Once
[40, 42, 41]	Numerical Solvers	Opt.	Instance	Single	Platform	Continuous	Mon/Params	Processing	Trigger
[81]	Planning	Both	Set	Single	Platform or Independent	Once or Continuous	Monolithic	Before processing	Once or Continuous
[97, 94, 96, 95]	Genetic Algorithms	Opt.	Instance	Set	Platform	Continuous	Mon/Params	Processing	Continuous
[114]	CSP	Dec.	Set	Set	Independent	Once	Monolithic	Design	Once
[116, 117]	Bayesian Networks	Opt.	Instance	Single	Independent	Once	Monolithic	Before processing	Once
[145]	CSP	Dec.	Both	Single	Independent	Once	Monolithic	Before processing	Once
[146, 147]	Numerical Solvers	Opt.	Instance	Single	Platform	Once or Triggered	Combinatorial	Before processing	Once
[149]	CSP	Opt.	Set	Set	Platform	Once	Monolithic	Before processing	Once
[188, 189]	Sorting	Opt.	Instance	Single	Machine	Continuous	Monolithic	Processing	Continuous
[193]	Sorting	Opt.	Instance	Set	Independent	Continuous	Generated	Compile or Run Time	Continuous
[204, 203, 205, 243]	Combinatorial Opt., SAT	Opt.	Instance	Both	Platform	Once	Monolithic	Before processing	Once
[206]	Sorting	Opt.	Instance	Single	Platform	Once	Mon/Params	Before processing	Once
[257]	Meta-Learning	Opt.	Instance	Single	Independent	Once	Monolithic	Before processing	Once
[270]	Numerical Solvers	Opt.	Instance	Single	Platform	Once	Monolithic	Before processing	Once
[311]	Network Communication (MPI)	Opt.	Instance	Single	Platform	Once	Mon/Params	Before and during processing	Continuous or Once
[317, 318, 319]	Matrix Multiplication	Opt.	Instance	Single	Machine	Once	Generated	Compile	Once
[338]	Simulation (Empirical Tuning)	Opt.	Instance	Single	Machine	Once	Monolithic	Before run	Once
[342]	Simulation (Empirical Tuning)	Opt.	Instance	Single	Machine	Once	Generated	Compile (and Processing)	Once (and Continuous)

Table A.1.: Categorization of the most relevant algorithm selection approaches (see fig. 2.13, p. 52). Only a subset of the categories is displayed, for clarity. In many other regards, the approaches are fairly similar — e.g., they all rely on empirical data. To keep the table as simple as possible, approaches from a single research group have been joined, as long as their categorization w.r.t. the above aspects is identical. The only simulation-related approaches focus on empirical tuning (sec. 2.7, p. 54), and are hence orthogonal to the developed methods (see fig. 2.15, p. 59).

A.4. Performance Database: Tables

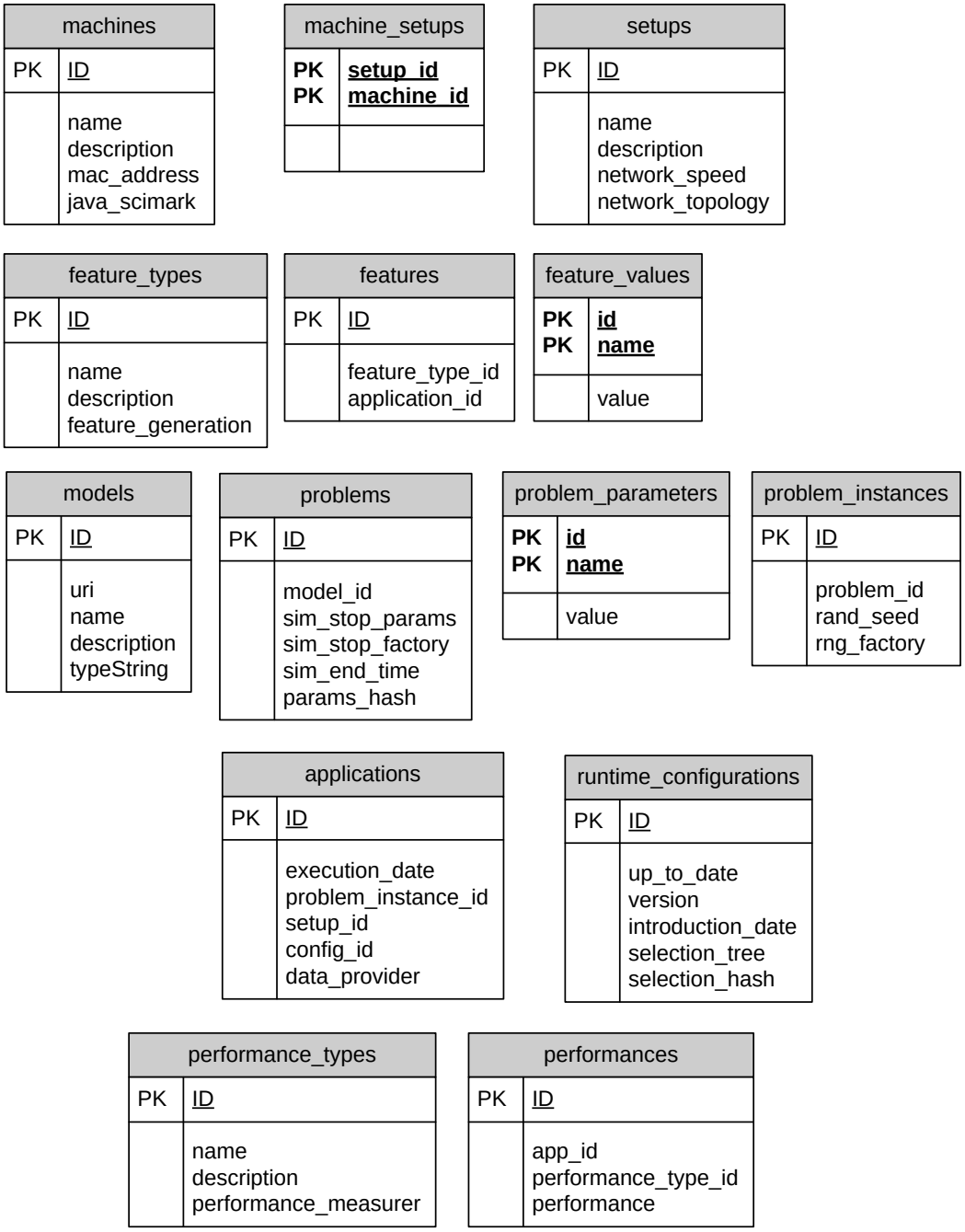

Figure A.1.: Tables of the performance database. Foreign-key relationships have been omitted for clarity. All tables are automatically created by Hibernate (see sec. 5.1.3, p. 111) .

A.5. Evaluating Simulation Algorithm Portfolio Selection with Synthetic Data

To get a broader picture on the performance gains that can be expected from pre-selecting simulation algorithm portfolios, a dedicated testbed for assessing portfolio selector effectiveness has been developed. The following description is a (slightly extended and adapted) excerpt from [75]. Section A.5.2 describes how synthetic test data for portfolio selection are generated, and section A.5.3 discusses experiments that show the (hypothetical) performance of different multi-armed bandit policies when their choice is restricted to the selected portfolios.

A.5.1. Portfolio Performance Metrics

The major challenge in evaluating portfolio selection for the adaptive simulation runner (sec. 7.2.1, p. 136) stems from the several levels on which stochastic effects occur. As already discussed in section 2.3.1 (p. 25), the execution time of an algorithm is influenced by various factors and should hence be regarded as a random variable. The performance of a multi-armed bandit policy (sec. 7.2.1, p. 136) depends on the stochastic performances of the algorithms — i.e., its rewards — and may itself exhibit random behavior (e.g., the ϵ policies). Moreover, the performance of portfolio selectors may be stochastic as well. For example, the selection mechanism based on genetic algorithms (sec. 7.2.2, p. 143) defines stochastic operations for selecting, recombining, and mutating the genomes of existing solutions (e.g., see fig. 7.12, p. 144).

The basic figure of merit is the *average execution time* $\theta(X, p_i, r, \alpha)$ of a simulation run when using policy X to execute r replications of simulation problem p_i with portfolio α:

$$\theta(X, p_i, r, \alpha) = \frac{\sum_{j=1}^{r} \hat{p}_{i,X(\alpha,j)}}{r} \tag{A.4}$$

where $X(\alpha, j)$ denotes the choice of policy X at the j-th round and $\hat{p}_{i,X(\alpha,j)}$ is a sample performance of the chosen algorithm on the given problem p_i, drawn from its runtime distribution for problem p_i.

The average execution times $\theta(X, p_i, r, \alpha)$ of a policy X can now be compared for *different* portfolios, e.g., retrieved from the GA-based selector, a random search, or just by including all algorithms into a portfolio. This allows to investigate the impact of portfolio selection on the performance of adaptive replication policies: the smaller the average execution time, the better. Since θ is a problem-specific metric, the execution times $\bar{\theta}(X, p_i, r, \alpha)$ are now averaged over a set $P \subseteq \mathbb{P}$ of simulation problems:

$$\Theta(\alpha, r, X) = \frac{\sum_{i=1}^{|P|} \bar{\theta}(X, p_i, r, \alpha)}{|P|} \tag{A.5}$$

where $\bar{\theta}$ denotes averaged calculations of equation A.4. Averaging is necessary because θ depends on randomly sampled algorithm performances and also potentially random policy behavior.

Finally, since portfolio selection may be stochastic *in itself*, we have to replicate the calculation of $\Theta(\alpha, r, X)$ with different portfolios α, selected on newly generated performance data with the same underlying correlations (see sec. A.5.2 below). The resulting *overall average execution time* $\bar{\Theta}$ is the quintessential performance indicator of using a portfolio approach to speed up adaptive replication with policy X. It denotes the expected average execution time of a simulation run when r replications are required.

A.5.2. Performance Data Generation

The synthetic performance data to test portfolio selection is generated by defining a number of *algorithm clusters*, each with the same number of members. All algorithms in a cluster are assumed to

behave similarly for the same kind of problem, but the standard deviations of their run time distributions differ.[1] Algorithm performances could be clustered because the corresponding selection trees all contain a plug-in that largely determines overall performance, e.g., a very efficient event queue.

The hypothetical problems are equally distributed over different *problem classes*. Each problem class can be solved best by algorithms from a specific cluster. Intuitively, the optimal portfolio contains the algorithm with the minimal performance variance for each cluster that is best for one problem class. Additional noise is introduced by associating a (randomized) hardness with each problem — some hypothetical simulation problems are hence generally easier to solve than others. Figure A.2 summarizes the overall setup.

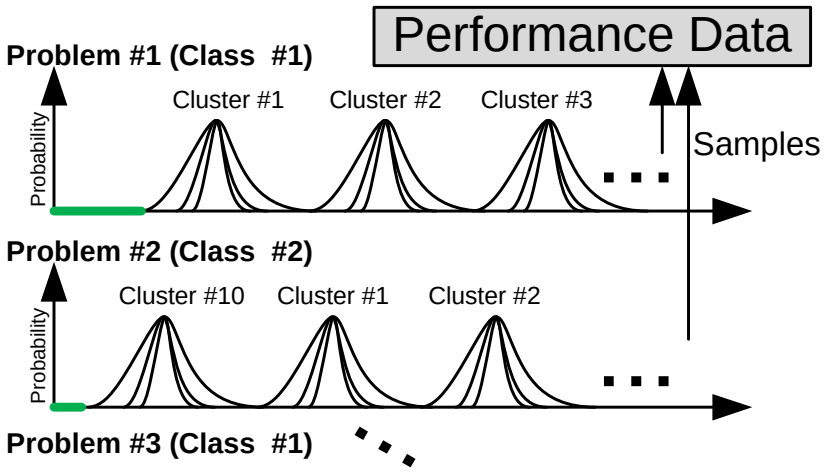

Figure A.2.: Generation of synthetic performance data. The problem hardness (green) is randomized, and the sequence of algorithm clusters depends on the problem class. Algorithms within a single cluster have run time distributions that differ in their standard deviations. The run time distributions of all algorithms are constructed as shown, for each problem. Then, the distributions are sampled to generate the synthetic performance data on which the portfolio selection operates (see fig. 7.13, p. 144).

A.5.3. Experiments

A genetic algorithm (GA) can be parameterized in various ways: the size of its population (γ_n), the number of generations to be computed (γ_g), and the mutation rate (γ_m) all affect solution quality and execution time. Furthermore, it relies on the recorded performance data (for the fitness function) and application-specific parameters like λ, the parameter to characterize the user's risk aversion (sec. 2.5.1, p. 39).

To systematically explore the effects of these aspects, a dedicated test environment generates some synthetic performance data as described above. All portfolio selectors under scrutiny are applied to the data, and their portfolios are then used in conjunction with the adaptive simulation runner. This allows to observe the effectiveness of each approach, also in comparison to not using portfolio selection at all (i.e., using a portfolio that contains all algorithms).

The adaptive simulation runner is configured with different policies and shall adaptively replicate $r = 200$ simulation runs. Its execution is replicated 20 times for each policy, to account for random algorithm performance and random policy behavior. The policies are assumed to replicate the simulation runs sequentially, i.e., they receive the feedback on one decision before they make the next one. The experiments consider the policies ϵ_n-Decreasing (ED), which was performing best in a previous

[1] If not stated otherwise, the run times are assumed to be normally distributed (see discussion in sec. 2.5.4, p. 45).

study [72], UCB2, and PURSUIT (P), all of which have been introduced in section 7.2.1 (p. 136).[2] Finally, a random selection policy (RS) is used to check the execution times without convergence to the most suitable algorithm.

All policies are compared on the same portfolios. New portfolios will be created from new synthetic data (with the same properties), and the whole procedure is repeated 10 times. This is done to estimate $\bar{\Theta}$ and its standard deviation, which results in $200 \cdot 20 \cdot 10 = 40,000$ hypothetical simulation runs per policy and $\bar{\Theta}$ value (see sec. A.5.1, p. 223). It is assumed that the synthetic performance data is *representative*. Hence, the algorithm performances $\hat{p}_{i,X(\alpha,j)}$ used for evaluation are drawn from the same probability distributions that are used to generate the data in the first place.

Synthetic performance data is generated for 100 simulation problems, each executed by 250 available algorithms that are assorted to 50 different clusters. Hence, this is a scenario where portfolio selection should be beneficial: there are more options (i.e., runtime configurations) than replications to be executed ($r = 200$). Synthetic performance data is gathered by averaging algorithm performance over three hypothetical executions. The execution time difference between two adjacent clusters is 10 s, i.e., algorithms from the fastest cluster will need just about 10 s to solve a simulation problem, whereas those from the slowest cluster need approximately 500 s. Similar numbers of algorithm combinations and a similar contrast between fastest and slowest algorithms have already been encountered in practice (e.g., [158]). There are 5 different classes of problems, each with another cluster being fastest.

For the genetic algorithm, mutation should not play too much of a role; its probability is set to $\gamma_m = 0,1\%$ (per available algorithm). The population size γ_n of the GA-based portfolio selector is set to 250, and the GA was executed for $\gamma_g = 400$ generations. Since execution times for portfolio selection were in the order of seconds for these scenarios (on an ordinary desktop computer) — whereas the execution times of the simulation replication experiments for which it is intended are in the order of minutes, hours, or even days — its overhead for the following scenarios is considered negligible and is not evaluated in more detail. To ensure a fair comparison, the stochastic portfolio selector (see sec. 7.2.2, p. 142) is configured to draw $\gamma_n \cdot \gamma_g = 100,000$ random portfolios. Portfolio size s is restricted to the interval $[3, 6]$.

A.5.4. Results

Effectiveness

As a first scenario, figure A.3 compares $\bar{\Theta}$ when using no portfolio selection (*None*), a portfolio selected by stochastic search (*Stochastic*), and a GA-selected portfolio (*GA*, configured with $\lambda = 1$, i.e., risky). Interestingly, UCB2 (which performs fairly average in the previous study [72]) benefits most and is able to outperform ϵ_n-DECREASING on pre-selected portfolios. It is slightly faster with portfolios selected by the GA than with those of the stochastic search. The bad performance of UCB2 in the absence of portfolio selection comes at no surprise, as it initially tries every option once (and hence has no chance to gain any speed-up in this scenario). In contrast, ϵ_n-DECREASING randomly explores the options and uses its best guess in-between, so it does much better without any portfolio. Finally, GA-selected portfolios exhibit less performance variance than portfolios selected by stochastic search, whereas the latter are suited for *all* kinds of policies: even in a risky setup they consistently outperform policy performance without portfolio selection. Still, they do not outperform GA-selection in peak performance (see UCB2 in fig. A.3).

Run Time Distributions

Since algorithm run times are not always distributed normally, it should be checked whether the performance improvement by portfolio selection is affected if execution times are distributed differently. The normal distribution was replaced by an Erlang distribution with $k = 2$ and $\theta = 2.0$, multiplied by 10% of the mean algorithm performance (to skew the distribution towards a longer tail). As

[2] Several other policies from the ϵ- and UCB-families have been evaluated as well, but the above policies were found to be representative.

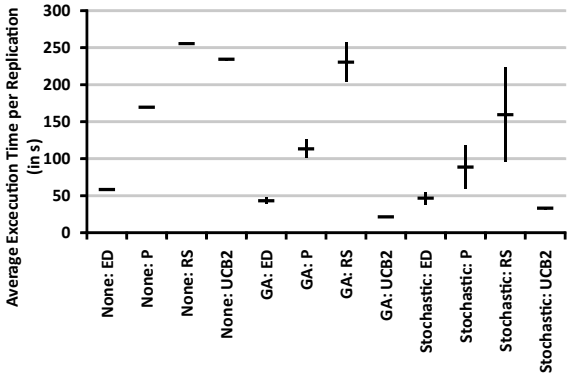

Figure A.3.: Average execution times in default scenario. As in the following, horizontal bars denote average values and vertical bars indicate the standard deviation.

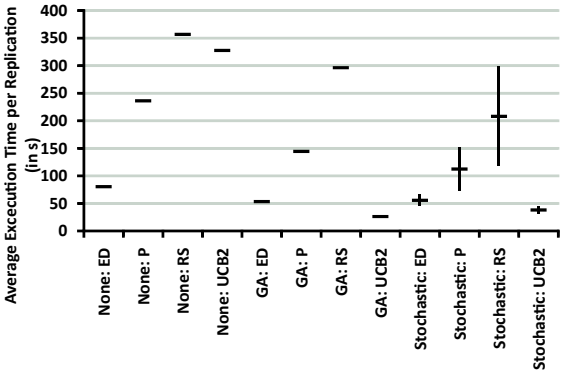

Figure A.4.: Similar scenario as in fig. A.3, but now with Erlang-distributed execution times.

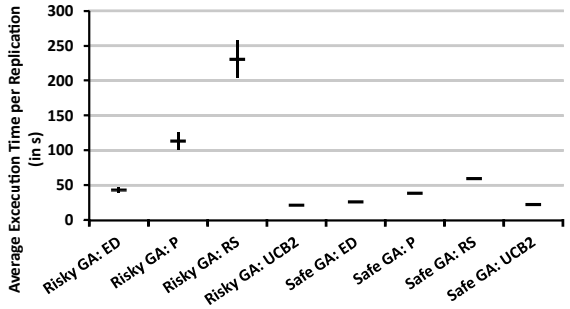

Figure A.5.: The impact of λ on the quality of GA-selected portfolios. Risky GA-selection was executed with $\lambda = 1$, safe GA-selection had λ set to 0.

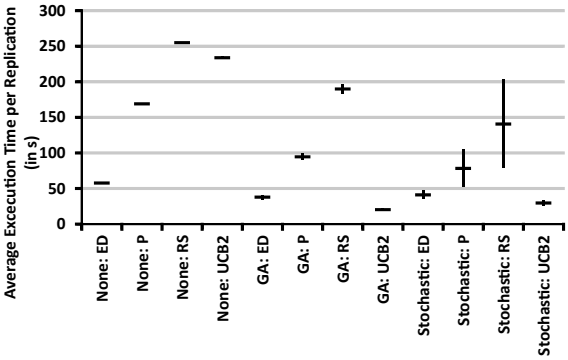

Figure A.6.: Similar scenario as in fig. A.3, but now only one replication is recorded.

figure A.4 shows, portfolio quality seems to be largely unaffected by runtime distribution — except for the performance variance of GA-selected portfolios, which is decreased. The results indicate that it should be rather safe to use GA-based portfolio selection, even though the runtime distributions might be unknown.

Risk

Another interesting aspect is the impact of λ on the quality of the selected portfolios. Figure A.5 compares the GA-selected portfolios generated with $\lambda = 1$ (from fig. A.3) and $\lambda = 0$. Adjusting the fitness function to take no risk ($\lambda = 0$) has the desired effect, i.e., even random algorithm selection now performs much better than without prior portfolio selection (cf. results for no portfolio selection in fig. A.3). This is useful in scenarios where adaptation cannot be guaranteed. UCB2 still outperforms the other policies: its performance on safe portfolios is only slightly worse than on risky ones.

Replications

Finally, it is important to evaluate the impact of noise in the performance data. If noisy data deteriorates the effectiveness of portfolio selection, it is necessary to eliminate the noise — e.g., by increasing the number of replications. The more replications are required per algorithm, the more time is required

for the prior simulation space exploration. Figure A.6 shows the performance gains of portfolio selection when each synthetic runtime distributions is sampled only once — instead of thrice — to construct the performance data. As the results are rather similar to those shown in figure A.3, it seems sufficient to try out each algorithm only once per problem. However, these results depend on the number of considered problems (noise may be balanced out) and the shape of the run time distributions.

A.6. Sample Listings

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugin xmlns="http://www.jamesii.org/plugin"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jamesii.org/plugin
    http://www.jamesii.org/plugin/plugin.xsd">
  <id name="Wumpusworld Simulator" version="1.0" />
  <factory classname="simulator.wumpusworld.WumpusWorldProcessorFactory"/>
</plugin>
```

Listing A.1: A sample plug-in definition file. It contains a unique name, a version, and the fully-qualified Java class name of the concrete factory.

```
<?xml version="1.0" encoding="UTF-8" ?>
<plugintype xmlns="http://www.jamesii.org/plugintype"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jamesii.org/plugintype
    http://www.jamesii.org/plugintype/plugintype.xsd">
  <id name="simulation/computation algorithm plug-ins" version="1.0" />
  <abstractfactory>james.core.processor.plugintype.AbstractProcessorFactory
  </abstractfactory>
  <basefactory>james.core.processor.plugintype.ProcessorFactory</basefactory>
  <description>Simulation/Computation algorithm plug-ins. A
      simulation/computation algorithm computes in a simulation run the
      trajectory of an initialized (executable) model. Typically
      simulation/computation algorithms will only work on one executable model
      interface, and thus there has to be at least one simulation algorithm
      per executable model "type".</description>
</plugintype>
```

Listing A.2: A sample plug-in type definition file. It contains a unique name and a version, as well as the fully-qualified class names of both abstract and base factory. It may also contain additional elements, such as a short description.

```
public class SampleWrapper implements ISampleJamesIIInterface {

  ExternalObject extObject;

  public SampleWrapper() {
    // Initialize external components
    // [...]
    extObject = new ExternalObject(...);
  }

  @Override
  public Result executeTask(ParameterBlock parameters) {
    // Convert James II parameters to parameters for the external component
    // [...]

    //Call the external component
    ExternalResult extResult = extObject.execTask();

    // Convert external result to a James II result object
    // [...]
    return new Result(...);
  }

}
```

Listing A.3: A sample wrapper, which implements a JAMES II interface and provides the functionality by internally delegating the task to an external component that is invisible from the outside.

```
public class BestSimulatorInTheWorldFactory extends ProcessorFactory {

  public SimulationRun create(IModel model, SimulationRun simulation,
      Partition partition, ParameterBlock params) {
    throw new RuntimeException("Error.");
  }

  public double getEfficencyIndex() {
    return Double.MAX_VALUE;
  }

  public List<Class<?>> getSupportedInterfaces() {
    List<Class<?>> result = new ArrayList<Class<?>>();
    result.add(IModel.class);
    return result;
  }

  public boolean supportsSubPartitions() {
    return true;
  }

}
```

Listing A.4: This processor factory (i.e., a factory to create a simulation algorithm) signals that the simulation algorithm it provides is applicable to *all* kinds of models (`getSupportedInterfaces()`) in JAMES II and even supports distributed simulation (`supportsSubPartitions()`). Since its efficiency of `Double.MAX_VALUE` cannot be surpassed, it will always be selected (unless automatic selection is overridden by a manual configuration).

Bibliography

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 103–112, Los Alamitos, CA, USA, April 2006. IEEE Computer Society.

[2] M. D. Adams, J. M. Kelley, J. D. Gocayne, M. Dubnick, M. H. Polymeropoulos, H. Xiao, C. R. Merril, A. Wu, B. Olde, R. F. Moreno, A. R. Kerlavage, W. R. McCombie, and J. C. Venter. Complementary DNA sequencing: expressed sequence tags and human genome project. *Science*, 252(5013):1651–1656, June 1991.

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

[4] Nitin Agrawal, Andrea C. Arpaci Dusseau, and Remzi H. Arpaci Dusseau. Towards realistic file-system benchmarks with codemri. *ACM SIGMETRICS Performance Evaluation Review*, 36(2):52–57, 2008.

[5] Robert Almeder. Pragmatism and science. In Stathis Psillos and Martin Curd, editors, *The Routledge Companion to Philosophy of Science*, Routledge Philosophy Companions, chapter 9, pages 91–99. Taylor & Francis, January 2008.

[6] Bruce Ankenman, Barry L. Nelson, and Jeremy Staum. Stochastic kriging for simulation metamodeling. In *Proceedings of the 40th Winter Simulation Conference*, pages 362–370. IEEE Computer Society, 2008.

[7] Jason Ansel, Cy Chan, Yee L. Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. *ACM SIGPLAN Notices*, 44(6):38–49, 2009.

[8] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 115–124. IEEE Conputer Society, 1999.

[9] A. Asuncion and D.J. Newman. UCI machine learning repository, 2007. http://archive.ics.uci.edu/ml/, Accessed 7/2010.

[10] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society, 1995.

[11] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[12] Vijay Balakrishnan, Peter Frey, Nael, and Philip A. Wilsey. A Framework for Performance Analysis of Parallel Discrete Event Simulators. In *Proceedings of the 29th Winter Simulation Conference*, pages 429–436. IEEE Computer Society, 1997.

[13] Bikramjit Banerjee, Ahmed Abukmail, and Landon Kraemer. Advancing the layered approach to agent-based crowd simulation. In *22nd Workshop on Principles of Advanced and Distributed Simulation (PADS 2008)*, pages 185–192, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[14] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation (3rd Edition)*. Prentice Hall, third edition, August 2000.

[15] Albert L. Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life*. Plume Books, April 2003.

[16] A. Basu, A. Bose, and J. K. Ghosh. An expository review of sequential designs and allocation rules. Technical Report 90-08, Department of Statistics, Purdue University, August 1990.

[17] David W. Bauer Jr, Christopher D. Carothers, and Akintayo Holder. Scalable Time Warp on Blue Gene supercomputers. In *23rd Workshop on Principles of Advanced and Distributed Simulation (PADS 2009)*, pages 35–44. IEEE, IEEE CPS, June 2009.

[18] Richard Bellman. A markovian decision process. *Indiana University Mathematics Journal*, 6:679–684, 1957.

[19] Michael W. Berry, Jack J. Dongarra, and Brian H. Larose. The development and implementation of a performance database server. Technical Report UT-CS-93-195, University of Tennessee, Knoxville, TN, USA, 1993.

[20] Michael Berthold and David J. Hand, editors. *Intelligent Data Analysis*. Springer, 1999.

[21] Maya Biersack, Viktor Friesen, Stefan Jähnichen, Matthias Klose, and Martin Simons. Towards an architecture for simulation environments. In Vren and Birta, editors, *Proceedings of the Summer Computer Simulation Conference (SCSC'95)*, pages 205–212. The Society for Computer Simulation, 1995.

[22] Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector D. Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. Reflection, self-awareness and self-healing in OpenORB. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 9–14, New York, NY, USA, 2002. ACM.

[23] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Concurrent replication of parallel and distributed simulations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS 2005)*, pages 234–243, 2005.

[24] Paul Bratley, Bennet L. Fox, and Linus E. Schrage. *A Guide to Simulation*. Springer, second edition, May 1987.

[25] Eric A. Brewer. High-level optimization via automated statistical modeling. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, New York, NY, USA, 1995. ACM Press.

[26] P. D. Bryan and T. M. Conte. Combining cluster sampling with single pass methods for efficient sampling regimen design. In *25th International Conference on Computer Design (ICCD 2007)*, pages 472–479. IEEE, 2007.

[27] Kevin Burrage, Tianhai Tian, and Pamela Burrage. A multi-scaled approach for simulating chemical reaction systems. *Progress in Biophysics and Molecular Biology*, 85(2-3):217–234, 2004.

[28] Hauke Busch, Werner Sandmann, and Verena Wolf. A numerical aggregation algorithm for the enzyme-catalyzed substrate conversion. In *Computational Methods in Systems Biology*, volume 4210/2006 of *Lecture Notes in Computer Science*, pages 298–311. Springer Berlin / Heidelberg, 2006.

[29] 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.

[30] Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Efficient step size selection for the tau-leaping simulation method. *The Journal of Chemical Physics*, 124(4), Jan 2006.

[31] Yang Cao, Hong Li, and Linda Petzold. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *The Journal of Chemical Physics*, 121(9):4059–4067, 2004.

[32] Luca Cardelli. Can a systems biologist fix a tamagotchi? http://lucacardelli.name/. Accessed 7/2010.

[33] Nicholas G. Carr. Is Google making us stupid? *The Atlantic Monthly*, July/August 2008.

[34] Francois E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, March 2006.

[35] Roger D. Chamberlain and Cheryl D. Henderson. Evaluating the use of pre-simulation in VLSI circuit partitioning. In *Proceedings of the eighth Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 139–146, New York, NY, USA, 1994. ACM.

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

[37] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of 12th International Joint Conference on AI (IJCAI-91)*, volume 1, pages 331–337, 1991.

[38] Dan Chen, Roland Ewald, Georgios K. Theodoropoulos, Robert Minson, Ton Oguara, Michael Lees, Brian Logan, and Adelinde M. Uhrmacher. Data access in distributed simulations of multi-agent systems. *Journal of Systems and Software*, 81(12):2345–2360, December 2008.

[39] Dan Chen, Stephen J. Turner, Boon P. Gan, Wentong Cai, Junhu Wei, and Nirupam Julka. Alternative solutions for distributed simulation cloning. *SIMULATION*, 79(5-6):299–315, May 2003.

[40] F. H. A. Claeys, P. A. Vanrolleghem, and P. Fritzson. A generalized framework for abstraction and dynamic loading of numerical solvers. In *Proceedings of the 2006 European Modeling and Simulation Symposium (EMSS)*, October 2006.

[41] Filip H. A. Claeys. *A generic software framework for modelling and virtual experimentation with complex biological systems*. PhD thesis, Ghent University, Dept. of Applied Mathematics, Biometrics and Process Control, January 2008.

[42] P. Claeys, F. Claeys, and P. A. Vanrolleghem. Intelligent configuration of numerical solvers of environmental ode/dae models using machine-learning techniques. In *Proceedings of the iEMSs 2006 Conference*. International Environmental Modelling and Software Society, July 2006.

[43] G. M. Constantinides and A. G. Malliaris. Portfolio theory. In R. A. Jarrow, V. Maksimovic, and W. T. Ziemba, editors, *Finance*, volume 9 of *Handbooks in OR & MS*, chapter 1. Elsevier, 1995.

[44] Thomas H. Cormen, Clifford Stein, Charles E. Leiserson, and Robert L. Rivest. *Introduction to Algorithms*. B&T, second revised edition edition, August 2001.

[45] Vittorio Cortellessa and Francesco Quaglia. An analysis of the efficiency of optimistically synchronized parallel simulators. In *Proceedings of the Conference on Simulation Methods and Applications*. Society for Computer Simulation, November 1998.

[46] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley Series in Telecommunications and Signal Processing. Wiley-Interscience, second edition, July 2006.

[47] Noel A. C. Cressie. *Statistics for Spatial Data*. Wiley Series in Probability and Statistics. Wiley, revised edition, January 1993.

[48] Eric Cronin, Anthony R. Kurc, Burton Filstrup, and Sugih Jamin. An efficient synchronization mechanism for mirrored game architectures. *Multimedia Tools and Applications*, 23(1):7–30, May 2004.

[49] Felipe Cucker and Ding X. Zhou. *Learning Theory: An Approximation Theory Viewpoint*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, May 2007.

[50] Olivier Dalle and Cyrine Mrabet. An instrumentation framework for component-based simulations based on the separation of concerns paradigm. In *Proceedings of the 6th EUROSIM Congress (EUROSIM'2007)*, September 2007.

[51] Olivier Dalle, Judicael Ribault, and Jan Himmelspach. Design considerations for m&s software. In M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, editors, *Proceedings of the 41st Winter Simulation Conference*, pages 944–955. IEEE Computer Society, 2009.

[52] Samir R. Das. Adaptive protocols for parallel discrete event simulation. In *Proceedings of the 28th Winter Simulation Conference*, pages 186–193, New York, NY, USA, 1996. ACM Press.

[53] Samir R. Das, Richard M. Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the 26th Winter Simulation Conference*, Proceedings of the 1994 Winter Simulation Conference, pages 1332 – 1339. Society for Computer Simulation International, 1994.

[54] Martin Davis, Ron Sigal, and Elaine J. Weyuker. *Computability, Complexity, and Languages, Second Edition : Fundamentals of Theoretical Computer Science*. Computer Science and Scientific Computing. Morgan Kaufmann, February 1994.

[55] Richard Dawkins. *The Selfish Gene*. Oxford University Press, 1976.

[56] D. Degenring, J. Lemcke, M. Röhl, and A. M. Uhrmacher. A variable structure model - the tryptophan operon. In G. Plotkin, editor, *Proceedings of the 3rd International Workshop on Computational Methods in Systems Biology*, pages 130–141, April 2005.

[57] Tom DeMarco. Software engineering: An idea whose time has come and gone? *IEEE Software*, 26(4):95–96, 2009.

[58] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 400(1818):97–117, 1985.

[59] Edsger W. Dijkstra. On the role of scientific thought. In David Gries, editor, *Selected Writings on Computing:A Personal Perspective*, Texts and Monographs in Computer Science, pages 60–66. Springer, 1982.

[60] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software for next generation applications. *International Journal of High Performance Computing Applications*, 17(2):125–131, May 2003.

[61] Bruce Edmonds. What is Complexity? - The philosophy of complexity per se with application to some examples in evolution. In F. Heylighen and D. Aerts, editors, *The Evolution of Complexity*. Kluwer, Dordrecht, 1999.

[62] J. Elf and M. Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *IET Systems Biology*, 1(2):230–236, Dec 2004.

[63] James Elliott, Ryan Fowler, and Tim O'Brien. *Harnessing Hibernate*. O'Reilly Media, first edition, May 2008.

[64] European Bioinformatics Institute. http://biomodels.net/. Accessed 7/2010.

[65] Roland Ewald. Simulation of Load Balancing Algorithms for Discrete Event Simulations. Master's thesis, University of Rostock, 2006.

[66] 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 Workshop on Principles of Advanced and Distributed Simulation (PADS 2006)*, 2006.

[67] Roland Ewald, Jan Himmelspach, Matthias Jeschke, Stefan Leye, and Adelinde M. Uhrmacher. Performance issues in evaluating models and designing simulation algorithms. In *Proceedings of the 2009 International Workshop on High Performance Computational Systems Biology*, pages 71–80. IEEE CPS, 2009.

[68] Roland Ewald, Jan Himmelspach, Matthias Jeschke, Stefan Leye, and Adelinde M. Uhrmacher. Flexible experimentation in the modeling and simulation framework JAMES II–implications for computational systems biology. *Briefings in Bioinformatics*, 11(3):290–300, May 2010.

[69] Roland Ewald, Jan Himmelspach, and Adelinde M. Uhrmacher. A non-fragmenting partitioning algorithm for hierarchical models. In *Proceedings of the 38th Winter Simulation Conference*, pages 848–855, Monterey, California, 2006. Winter Simulation Conference.

[70] Roland Ewald, Jan Himmelspach, and Adelinde M. Uhrmacher. An algorithm selection approach for simulation systems. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS 2008)*, volume 22, pages 91–98, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[71] Roland Ewald, Jan Himmelspach, Adelinde M. Uhrmacher, Dan Chen, and Georgios K. Theodoropoulos. A simulation approach to facilitate parallel and distributed discrete-event simulator development. In *Proceedings of the 10th IEEE International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2006*, pages 209–218, Washington, DC, USA, 2006. IEEE Computer Society.

[72] Roland Ewald, Stefan Leye, and Adelinde M Uhrmacher. An efficient and adaptive mechanism for parallel simulation replication. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS 2009)*, pages 104–113. IEEE CPS, 2009.

[73] Roland Ewald, Carsten Maus, Arndt Rolfs, and Adelinde M. Uhrmacher. Discrete event modeling and simulation in systems biology. *Journal of Simulation*, 1(2):81–96, 2007.

[74] Roland Ewald, Johannes Rössel, Jan Himmelspach, and Adelinde M. Uhrmacher. A plug-in - based architecture for random number generation in simulation systems. In S. J. Mason, R. R. Hill, L. Moench, and O. Rose, editors, *Proceedings of the 40th Winter Simulation Conference*, 2008.

[75] Roland Ewald, René Schulz, and Adelinde M. Uhrmacher. Selecting simulation algorithm portfolios by genetic algorithms. In *IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 48–56, Piscataway, NJ, May 2010. IEEE CPS.

[76] Roland Ewald, Adelinde Uhrmacher, and Kaustav Saha. Data mining for simulation algorithm selection. In *Proceedings of the SIMUTools'09: 2nd International Conference on Simulation Tools and Techniques*. ICST, 2009.

[77] Roland Ewald and Adelinde M. Uhrmacher. Automating the runtime performance evaluation of simulation algorithms. In M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, editors, *Proceedings of the 41st Winter Simulation Conference*, pages 1079–1091. IEEE Computer Society, 2009.

[78] Michael Fellows. Parameterized complexity: The main ideas and some research frontiers. *Algorithms and Computation*, pages 291–307, 2001.

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

[80] Richard P. Feynman and Tony Hey. *Feynman Lectures on Computation*. Perseus Books Group, August 1996.

[81] Eugene Fink. How to solve it automatically: Selection among problem solving methods. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 128–136. AAAI Press, 1998.

[82] Ulrich Finkler and Kurt Mehlhorn. Runtime prediction of real programs on real machines. In *SODA '97: Proceedings of the eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 380–389, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[83] Per O. Fjällström. Algorithms for graph partitioning: A Survey. *Linköping Electronic Articles in Computer and Information Science*, 3(98-010), 1998.

[84] Malcolm Forster. Prediction. In Stathis Psillos and Martin Curd, editors, *The Routledge Companion to Philosophy of Science*, The Routledge Companion to Philosophy of Science, chapter 38, pages 405–413. Taylor & Francis, January 2008.

[85] Ian Foster and Carl Kesselman. Globus: a metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, June 1997.

[86] Martin Fowler and Scott Kendall. *UML Distilled: Applying the Standard Object Modelling Language*. Addison-Wesley, August 1997.

[87] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384, 1998.

[88] M. C. Fu, F. W. Glover, and J. April. Simulation optimization: a review, new developments, and applications. In *Proceedings of the 2005 Winter Simulation Conference*, 2005.

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

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

[91] Richard M. Fujimoto. Parallel simulation: distributed simulation systems. In *Proceedings of the 35th Winter Simulation Conference*, pages 124–134. Winter Simulation Conference, 2003.

[92] Richard M. Fujimoto. Modeling, simulation, and parallel computation: The future is now. Keynote Address SpringSim'07, March 2007.

[93] Matteo Gagliolo and Jürgen Schmidhuber. A neural network model for inter-problem adaptive online time allocation. In *Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005*, Lecture Notes in Computer Science, pages 7–12. Springer Berlin / Heidelberg, 2005.

[94] Matteo Gagliolo and Jürgen Schmidhuber. Towards life-long meta learning. Poster paper at Workshop 'Inductive Transfer : 10 Years Later' (Nineteenth Annual Conference on Neural Information Processing Systems, NIPS 2005), December 2005.

[95] Matteo Gagliolo and Jürgen Schmidhuber. Dynamic algorithm portfolios. In *Ninth International Symposium on Artificial Intelligence and Mathematics (AI & MATH '06)*, January 2006.

[96] Matteo Gagliolo and Jürgen Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, August 2006.

[97] Matteo Gagliolo, Viktor Zhumatiy, and Jürgen Schmidhuber. Adaptive online time allocation to search algorithms. In J. F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, editors, *Machine Learning: ECML 2004. Proceedings of the 15th European Conference on Machine Learning*, Lecture Notes in Computer Science, pages 134–143. Springer Berlin / Heidelberg, September 2004.

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

[99] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. Developing component architectures for distributed scientific problem solving. *IEEE Computational Science & Engineering*, 5(2):50–63, 1998.

[100] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *STOC '74: Proceedings of the sixth Annual ACM Symposium on Theory of Computing*, pages 47–63, New York, NY, USA, 1974. ACM Press.

[101] Ian P. Gent, Stuart A. Grant, Ewen MacIntyre, Patrick Prosser, Paul Shaw, Barbara M. Smith, and Toby Walsh. How not to do it. Technical Report 97.27, University of Leeds, May 1997.

[102] James E. Gentle. *Numerical Linear Algebra for Applications in Statistics*. Springer, first edition, August 1998.

[103] Daniele Gianni, Giuseppe Iazeolla, and Andrea D'Ambrogio. A methodology to predict the performance of distributed simulations. In *IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 31–39, Piscataway, NJ, May 2010. IEEE CPS.

[104] Michael A. Gibson and Jehoshua Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *The Journal of Chemical Physics*, 104:1876–1889, 2000.

[105] Daniel T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Computational Physics*, 22(4):403–434, December 1976.

[106] Daniel T. Gillespie. Exact Stochastic Simulation of Coupled Chemical Reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[107] Daniel T. Gillespie. A rigorous derivation of the chemical master equation. *Physica A: Statistical Mechanics and its Applications*, 188(1-3):404–425, September 1992.

[108] Daniel T. Gillespie. Stochastic simulation of chemical kinetics. *Annual review of physical chemistry*, 58(1):35–55, 2007.

[109] John C. Gittins. *Multi-Armed Bandit Allocation Indices*. Wiley Interscience Series in Systems and Optimization. John Wiley and Sons Ltd, January 1989.

[110] E. Glinsky and G. Wainer. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Ninth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT 2005)*, pages 265–272, 2005.

[111] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, May 2006.

[112] Rick S. Goh and Ian L. Thng. Mlist: An efficient pending event set structure for discrete event simulation. *International Journal of Simulation - Systems, Science & Technology*, 4(5-6):66–77, December 2003.

[113] Carla P. Gomes and Bart Selman. Algorithm portfolio design: Theory vs. practice. In Dan Geiger and Prakash P. Shenoy, editors, *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI-97)*, pages 190–197, San Francisco, 1997. Morgan Kaufmann Publishers.

[114] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, February 2001.

[115] Peter Grassberger. On correlations in "good" random number generators. *Physics Letters A*, 181(1):43–46, September 1993.

[116] Haipeng Guo. *Algorithm Selection for Sorting and Probabilistic Inference: A Machine-Learning Approach*. PhD thesis, Kansas State University, 2003.

[117] Haipeng Guo and William Hsu. A machine learning approach to algorithm selection for NP-hard optimization problems: a case study on the MPE problem. *Annals of Operations Research*, 156(1):61–82, December 2007.

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

[119] Mads Haahr. http://www.random.org/. Accessed 7/2010.

[120] Lou Hafer and Arthur E. Kirkpatrick. Assessing open source software as a scholarly contribution. *Communications of the ACM*, 52(12):126–129, 2009.

[121] Frank Hampel. Robust statistics: A brief introduction and overview. Technical Report 94, Eidgenössische Technische Hochschule (ETH) Zürich, March 2001.

[122] T. Hastie, R. Tibshirani, and J. H. Friedman. *The Elements of Statistical Learning*. Springer, August 2001.

[123] P. Heidelberger and S. S. Lavenberg. Computer performance evaluation methodology. *IEEE Transactions on Computers*, C-33(12):1195–1220, 1984.

[124] Philip Heidelberger. Statistical analysis of parallel simulations. In *Proceedings of the 18th Winter Simulation Conference*, pages 290–295, New York, NY, USA, 1986. ACM.

[125] P. Hellekalek. Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, 46(5-6):485–505, 1998.

[126] Robert Henderson. Job scheduling under the portable batch system. In *Job Scheduling Strategies for Parallel Processing*, volume 949/1995 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin / Heidelberg, 1995.

[127] S. G. Henderson and B. L. Nelson, editors. *Handbooks in Operations Research and Management Science, Volume 13: Simulation*. North Holland, first edition, November 2006.

[128] Francis Heylighen. Principles of systems and cybernetics: an evolutionary perspective. In Robert Trappl, editor, *Proceedings of the eleventh European Meeting on Cybernetics and Systems Research*, pages 3–10. World Scientific Singapore, 1992.

[129] Hibernate. https://www.hibernate.org/. Accessed 7/2010.

[130] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[131] Jan Himmelspach. *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations- und Experimentiersystems*. PhD thesis, University of Rostock, November 2007.

[132] Jan Himmelspach, Roland Ewald, and Adelinde M Uhrmacher. A flexible and scalable experimentation layer. In S.J. Mason, R.R. Hill, L. Moench, and O. Rose, editors, *Proceedings of the 40th Winter Simulation Conference*, 2008.

[133] Jan Himmelspach and Adelinde M. Uhrmacher. A component-based simulation layer for JAMES. In *Proceedings of the 18th Workshop on Parallel and Distributed Simulation (PADS '04)*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[134] Jan Himmelspach and Adelinde M. Uhrmacher. Sequential processing of PDEVS models. In Agostino G. Bruzzone, Antoni Guasch, Miquel A. Piera, and Jerzy Rozenblit, editors, *Proceedings of the 3rd EMSS*, pages 239–244, October 2006.

[135] Jan Himmelspach and Adelinde M. Uhrmacher. The event queue problem and PDEVS. In *Proceedings of the SpringSim '07, DEVS Integrative M&S Symposium*, pages 257–264. SCS, 2007.

[136] Jan Himmelspach and Adelinde M. Uhrmacher. Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium*, pages 137–143. IEEE Computer Society, 2007.

[137] Vlatka Hlupic. Discrete-event simulation software: What the users want. *SIMULATION*, 73(6):362–370, December 1999.

[138] Roger W. Hockney. A framework for benchmark performance analysis. *Supercomputer*, 9(2):9–22, March 1992.

[139] Tad Hogg, Bernardo A. Huberman, and Colin P. Williams. Phase transitions and the search problem. *Artificial Intelligence*, 81(1-2):1–15, March 1996.

[140] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, April 1992.

[141] John H. Holland. *Hidden Order: How Adaptation Builds Complexity (Helix Books)*. Addison Wesley Publishing Company, June 1996.

[142] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42(2):201–212, 1994.

[143] Holger H. Hoos and Thomas Stützle. SATLIB: an online resource for research on sat. In Ian P. Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000: highlights of satisfiability research in the year 2000*, Frontiers in Artificial Intelligence and Applications, pages 283–292. IOS Press, 2000.

[144] Mark F. Hornick, Erik Marcade, and Sunil Venkayala. *Java Data Mining: Strategy, Standard, and Practice. A Practical Guide for Architecture, Design, and Implementation.* Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2007.

[145] Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry A. Kautz, Bart Selman, and David M. Chickering. A bayesian approach to tackling hard computational problems. In *UAI '01: Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence*, pages 235–244, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[146] E. N. Houstis, A. Catlin, J. Rice, V. Verykios, N. Ramakrishnan, and C. Houstis. PYTHIA II: a knowledge/database system for managing performance data and recommending scientific software. *ACM Transactions on Mathematical Software*, 26(2):227–253, 2000.

[147] E. N. Houstis, A. C. Catlin, N. Dhanjani, J. R. Rice, N. Ramakrishnan, and V. Verykios. MyPYTHIA: a recommendation portal for scientific software and services. *Concurrency and Computation: Practice and Experience*, 14(13-15):1481–1505, 2002.

[148] Elias N. Houstis and John R. Rice. Future problem solving environments for computational science. In Ronald F. Boisvert and Elias N. Houstis, editors, *Computational science, mathematics and software*, pages 93–114. Purdue University Press, West Lafayette, IN, USA, 2002.

[149] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.

[150] K. A. Huck and A. D. Malony. PerfExplorer: A performance data mining framework for Large-Scale parallel computing. In *Proceedings of the ACM/IEEE SC 2005 Conference*, pages 41–52, 2005.

[151] Kevin A. Huck, Allen D. Malony, Robert Bell, and Alan Morris. Design and implementation of a parallel performance data management framework. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 473–482, Washington, DC, USA, 2005. IEEE Computer Society.

[152] Maria Hybinette and Richard M. Fujimoto. Cloning parallel simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, 2001.

[153] IBM. Autonomic computing: IBM's perspective on the state of information technology, 2001. www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, Accessed 7/2010.

[154] John P. A. Ioannidis. Why most published research findings are false. *PLoS Medicine*, 2(8):696–701, August 2005.

[155] Engin Ipek, Sally A. Mckee, Karan Singh, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficient architectural design space exploration via predictive modeling. *ACM Transactions on Architecture and Code Optimization*, 4(4):1–34, January 2008.

[156] Nobuyoshi Ishii, Tomoyoshi Soga, Takaaki Nishioka, and Masaru Tomita. Metabolome analysis and metabolic simulation. *Metabolomics*, 1(1):29–37, March 2005.

[157] 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.

[158] Matthias Jeschke and Roland Ewald. Large-scale design space exploration of SSA. In *Computational Methods in Systems Biology (CMSB 2008)*, volume 5307 of *Lecture Notes in Computer Science*, pages 211–230. Springer Berlin / Heidelberg, October 2008.

[159] Matthias Jeschke, Roland Ewald, Alfred Park, Richard Fujimoto, and Adelinde M. Uhrmacher. A parallel and distributed discrete event approach for spatial cell-biological simulations. *ACM SIGMETRICS Performance Evaluation Review*, 35(4):22–31, March 2008.

[160] Matthias Jeschke, Alfred Park, Roland Ewald, Richard Fujimoto, and Adelinde M. Uhrmacher. Parallel and distributed spatial simulation of chemical reactions. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS '08)*, pages 51–59, Washington, DC, USA, 2008. IEEE Computer Society.

[161] Matthias Jeschke and Adelinde M. Uhrmacher. Multi-resolution spatial simulation for molecular crowding. In S. J. Mason, R. R. Hill, L. Moench, and O. Rose, editors, *Proceedings of the 40th Winter Simulation Conference*, December 2008.

[162] Mathias John, Roland Ewald, and Adelinde M. Uhrmacher. A spatial extension to the [pi] calculus. *Electronic Notes in Theoretical Computer Science*, 194(3):133–148, January 2008.

[163] D. Johnson. A theoretician's guide to the experimental analysis of algorithms. In Michael Goldwasser, David Johnson, and Catherine McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges : Papers Related to the DIMACS Challenge on Dictionaries and Priority Queues (1995-1996) and the DIMACS Challenge on Near Neighbor Searches (1998-1999)*, volume 59 of *DIMACS series in discrete mathematics and theoretical computer science*, pages 215–250. American Mathematical Society, Providence, RI, United States, December 2002.

[164] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[165] Philippe Jorion. Portfolio optimization in practice. *Financial Analysts Journal*, 48(1):68–74, January 1992.

[166] JSR-247 Expert Group. Java(tm) Specification Request 247: Java^{TM}Data Mining (JDM) 2.0, September 2006.

[167] JSR-73 Expert Group. Java(tm) Specification Request 73: Java^{TM}Data Mining (JDM), July 2004.

[168] 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.

[169] 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*, 2002.

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

[171] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[172] Leslie P. Kaelbling. *Learning in Embedded Systems*. MIT Press, May 1993.

[173] Boyko Kakaradov. Ultra-fast matrix multiplication: An empirical analysis of highly optimized vector algorithm. *Stanford Undergraduate Research Journal (SURJ)*, 3:33–36, 2004.

[174] Alexandros Kalousis, João Gama, and Melanie Hilario. On data and algorithms: Understanding inductive performance. *Machine Learning*, 54(3):275–312, March 2004.

[175] Poul H. Kamp. You're doing it wrong. *Queue*, 8(6):20–27, 2010.

[176] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gabor Peceli, Gyula Simon, and Tamas Kovacshazy. An approach to self-adaptive software based on supervisory control. In *Self-Adaptive Software: Applications*, volume 2614/2003 of *Lecture Notes in Computer Science*, pages 77–92. Springer Berlin / Heidelberg, 2003.

[177] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003.

[178] Ross D. King, Jem Rowland, Stephen G. Oliver, Michael Young, Wayne Aubrey, Emma Byrne, Maria Liakata, Magdalena Markham, Pinar Pir, Larisa N. Soldatova, Andrew Sparkes, Kenneth E. Whelan, and Amanda Clare. The automation of science. *Science*, 324(5923):85–89, April 2009.

[179] Akira R. Kinjo and Shoji Takada. Competition between protein folding and aggregation with molecular chaperones in crowded solutions: Insight from mesoscopic simulations. *Biophysical Journal*, 85(6):3521–3531, December 2003.

[180] H. Kitano. Systems biology: a brief overview. *Science*, 295(5560):1662–1664, 2002.

[181] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[182] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, 1976.

[183] Donald E. Knuth. *Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley Professional, second edition, November 1981.

[184] R. Kohavi, G. John, R. Long, D. Manley, and K. Pfleger. MLC++: a machine learning library in C++. In *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, pages 740–743, 1994.

[185] Dagmar Köhn and Nicolas Le Novère. SED-ML - an XML format for the implementation of the MIASE guidelines. In *Computational Methods in Systems Biology (CMSB 2008)*, volume 5307 of *Lecture Notes in Computer Science*, pages 176–190. Springer Berlin / Heidelberg, 2008.

[186] KXen Inc. http://www.kxen.com. Accessed 7/2010.

[187] R. Laddaga. Creating robust software through self-adaptation. *IEEE Intelligent Systems and their Applications*, 14(3):26–29, 1999.

[188] Michail G. Lagoudakis and Michael L. Littman. Algorithm selection using reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*, pages 511–518. Morgan Kaufmann, San Francisco, CA, 2000.

[189] Michail G. Lagoudakis, Michael L. Littman, and Ronald E. Parr. Selecting the right algorithm. In *In Proceedings of the 2001 AAAI Fall Symposium Series: Using Uncertainty within Computation*, 2001.

[190] T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6(1):4–22, March 1985.

[191] Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *SODA '97: Proceedings of the eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[192] Averill Law. *Simulation Modeling and Analysis*. McGraw-Hill Publishing Co., fourth edition, August 2006.

[193] A. Laws, A. Taleb-Bendiab, and S. Wade. Genetically modified software: Realizing viable autonomic agency. In *Innovative Concepts for Autonomic and Agent-Based Systems*, volume 3825/2006 of *Lecture Notes in Computer Science*, pages 184–196. Springer Berlin / Heidelberg, 2006.

[194] A. Laws, A. Taleb-Bendiab, S. Wade, and D. Reilly. From wetware to software: A cybernetic perspective of self-adaptive software. In *Self-Adaptive Software: Applications*, volume 2614/2003 of *Lecture Notes in Computer Science*, pages 341–357. Springer Berlin / Heidelberg, 2003.

[195] Pierre L'Ecuyer. Random numbers for simulation. *Commununications of the ACM*, 33(10):85–97, 1990.

[196] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4), August 2007.

[197] Ethan Lee, Adrian Salic, Roland Krüger, Reinhart Heinrich, and Marc W. Kirschner. The roles of APC and Axin derived from experimental and theoretical analysis of the Wnt pathway. *PLoS Biology*, 1(1):e10, 2003.

[198] Michael Lees, Brian Logan, Chen Dan, Ton Oguara, and Georgios K. Theodoropoulos. Decision-Theoretic Throttling for Optimistic Simulations of Multi-Agent Systems. In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2005*, pages 171–178, 2005.

[199] 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*, 2004.

[200] Stefan Leye, Mathias John, and Adelinde M. Uhrmacher. A flexible architecture for performance experiments with the pi-calculus and its extensions. In *3rd International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS'2010), Malaga, Spain*, March 2010.

[201] Stefen Leye, Jan Himmelspach, Matthias Jeschke, Roland Ewald, and Adelinde M. Uhrmacher. A grid-inspired mechanism for coarse-grained experiment execution. In *Proceedings of the 12th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2008*, pages 7–16. IEEE Computer Society, 2008.

[202] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. Boosting as a metaphor for algorithm design. In *ICCP: International Conference on Constraint Programming (CP)*, volume 2833/2003 of *Lecture Notes in Computer Science*, pages 899–903. Springer Berlin / Heidelberg, 2003.

[203] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. A portfolio approach to algorithm selection. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1542–1543. Morgan Kaufmann Publishers Inc., 2003.

[204] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning and empirical hardness of optimization problems: The case of combinatorial auctions. In *ICCP: International Conference on Constraint Programming (CP), LNCS*, volume 2470/2006 of *Lecture Notes in Computer Science*, pages 91–100. Springer Berlin / Heidelberg, 2002.

[205] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4):1–52, June 2009.

[206] X. Li, M. J. Garzarán, and D. Padua. A dynamically tuned sorting library. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 111–122. IEEE Computer Society, 2004.

[207] Xiaoming Li and María J. Garzarán. Optimizing matrix multiplication with a classifier learning system. In *Languages and Compilers for Parallel Computing*, volume 4339/2006 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin / Heidelberg, 2006.

[208] Jason Liu, David M. Nicol, Brian J. Premore, and Anna L. Poplawski. Performance Prediction of a Parallel Simulator. In *Proceedings of the thirteenth Workshop on Parallel and Distributed Simulation (PADS '99)*, pages 156–164, 1999.

[209] Qi Liu and Gabriel Wainer. Exploring multi-grained parallelism in compute-intensive DEVS simulations. In *IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 65–72, Piscataway, NJ, May 2010. IEEE CPS.

[210] Andrew W. Lo and A. Craig MacKinlay. Stock market prices do not follow random walks: Evidence from a simple specification test. *The Review of Financial Studies*, 1(1):41–66, 1988.

[211] Brian Logan and Georgios K. Theodoropoulos. The distributed simulation of multi-agent systems. *Proceedings of the IEEE*, 89(2):174–186, February 2001.

[212] M. Y. H. Low. Dynamic load-balancing for BSP time warp. In *Proceedings of the 35th Annual Simulation Symposium*, pages 267–274, 2002.

[213] Elizabeth Lynch and George Riley. Hardware supported time synchronization in multi-core architectures. In *23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*. IEEE, IEEE CPS, June 2009.

[214] Machine Learning in Java. http://sourceforge.net/projects/mldev/. Accessed 7/2010.

[215] Harry Markowitz. Portfolio selection. *The Journal of Finance*, 7(1):77–91, 1952.

[216] Paolo Marrone. *Java Object Oriented Neural Engine: The Complete Guide*, January 2007.

[217] George Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 61(1):25–28, September 1968.

[218] George Marsaglia. Seeds for random number generators. *Communications of the ACM*, 46(5):90–93, May 2003.

[219] Alke Martens, Jan Himmelspach, and Roland Ewald. Modeling, simulation and games. In U. Lucke, M. C. Kindsmüller, S. Fischer, M. Herczeg, and S. Seehusen, editors, *Workshop Proceedings der Tagungen Mensch und Computer 2008, Delfi 2008 und Cognitive Design 2008*, pages 349–354. GI, Logos Verlag, Berlin, 2008.

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

[221] 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 *Proceedings of the 36th Annual Symposium on Simulation*, pages 216–223. IEEE Computer Society, 2003.

[222] Michael Mascagni and Ashok Srinivasan. Parameterizing parallel multiplicative lagged-Fibonacci generators. *Parallel Computing*, 30(7):899–916, July 2004.

[223] MathWorks. http://www.mathworks.com/. Accessed 7/2010.

[224] MathWorks. http://www.mathworks.com/products/simulink/. Accessed 7/2010.

[225] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[226] Makoto Matsumoto, Isaku Wada, Ai Kuramoto, and Hyo Ashihara. Common defects in initialization of pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 17(4), September 2007.

[227] Catherine McGeoch. Analyzing algorithms by simulation: variance reduction techniques and simulation speedups. *ACM Computing Surveys*, 24(2):195–212, 1992.

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

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

[230] Philip K. Mckinley, Seyed M. Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[231] Sina Meraji, Wei Zhang, and Carl Tropper. A multi-state Q-learning approach for the dynamic load balancing of time warp. In *IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 142–149, Piscataway, NJ, May 2010. IEEE CPS.

[232] Rob Minson and Georgios Theodoropoulos. Adaptive support of range queries via push-pull algorithms. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS '07)*, pages 53–60, Washington, DC, USA, 2007. IEEE Computer Society.

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

[234] Gautam Mitra, Triphonas Kyriakis, Cormac Lucas, and Mehndi Pirbhai. A review of portfolio planning: models and systems. In Stephen Satchell and Alan Scowcroft, editors, *Advances in Portfolio Construction and Implementation*, Quantitative Finance Series, chapter 1, pages 1–39. Butterwoth-Heinemann, June 2003.

[235] Bernard M. E. Moret. Towards a discipline of experimental algorithmics. In Michael Goldwasser, David Johnson, and Catherine McGeoch, editors, *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges : Papers Related to the DIMACS Challenge on Dictionaries and Priority Queues (1995-1996) and the DIMACS Challenge on Near Neighbor Searches (1998-1999)*, pages 197–213. American Mathematical Society, Oxford, United States, February 2003.

[236] MySQL. http://www.mysql.com/. Accessed 7/2010.

[237] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Communications of the ACM*, 19(3):113–126, March 1976.

[238] 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.

[239] David M. Nicol. Utility analysis of parallel simulation. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation (PADS '03)*, Washington, DC, USA, 2003. IEEE Computer Society.

[240] David M. Nicol, Michael M. Johnson, Ann S. Yoshimura, and Michael E. Goldsby. Performance Modeling of the IDES Framework. In *Proceedings of the eleventh Workshop on Parallel and Distributed Simulation (PADS '97)*, pages 38–45, 1997.

[241] David M. Nicol, Jason Liu, Michael Liljenstam, and Guanhua Yan. Simulation of large-scale networks using SSF. In *Proceedings of the 35th Winter Simulation Conference*, pages 650–657, 2003.

[242] Denis Noble. *The Music of Life: Biology beyond the Genome*. Oxford University Press, USA, June 2006.

[243] Eugene Nudelman, Kevin Leyton-Brown, Alex Devkar, Yoav Shoham, and Holger Hoos. SATzilla: An Algorithm Portfolio for SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing, SAT 2004 Competition: Solver Descriptions*, pages 13–14, 2004.

[244] 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 *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT 2005)*, pages 179–186, 2005.

[245] Oracle Corporation. http://download.oracle.com/javase/tutorial/reflect/. Accessed 9/2011.

[246] Oracle Corporation. http://java.sun.com/javase/technologies/database/. Accessed 7/2010.

[247] E. H. Page, D. M. Nicol, O. Balci, R. M. Fujimoto, P. A. Fishwick, P. L'Ecuyer, and R. Smith. Panel: strategic directions in simulation research. In *Proceedings of the 1999 Winter Simulation Conference*, volume 2, pages 1509–1520 vol.2, 1999.

[248] Christos Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, 12(3):441–450, 1987.

[249] Dirk Pawlaszczyk and Steffen Strassburger. Scalability in distributed simulations of agent-based models. In M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, editors, *Proceedings of the 41st Winter Simulation Conference*, pages 1189–1200. IEEE Computer Science, December 2009.

[250] K. Pawlikowski, H. D. J. Jeong, and J. S. R. Lee. On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine*, 40(1):132–139, 2002.

[251] C. Peng. Parallel discrete event simulation of manufacturing systems: A technology survey. *Computers & Industrial Engineering*, 31(1-2):327–330, October 1996.

[252] ACM SIGMETRICS Performance Evaluation Review. ACM Journal: http://www.sigmetrics.org/per.shtml (Accessed 4/2010).

[253] Kalyan S. Perumalla, Brandon G. Aaby, Srikanth B. Yoginath, and Sudip K. Seal. GPU-based real-time execution of vehicular mobility models in large-scale road network scenarios. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*, pages 95–103, Washington, DC, USA, June 2009. IEEE Computer Society.

[254] 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 *Proceedings of the 37th Winter Simulation Conference*, 2005.

[255] Erez Petrank and Dror Rawitz. The hardness of cache conscious data placement. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 101–112, New York, NY, USA, 2002. ACM.

[256] Marek Petrik. Statistically optimal combination of algorithms. In *Local Proceedings of SOFSEM 2005*, January 2005.

[257] Bernhard Pfahringer, Hilan Bensusan, and Christophe Giraud-Carrier. Meta-learning by landmarking various learning algorithms. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pages 743–750, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[258] Joseph P. Pickett, David R. Pritchard, and Christopher Leonesia, editors. *The American Heritage Dictionary*. Bantam Dell, fourth edition, 2004.

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

[260] Karl Popper. Selections from 'the logic of scientific discovery'. In Richard Boyd, Philip Gasper, and J. D. Trout, editors, *The philosophy of science*, chapter 5, pages 99–119. The MIT Press, June 1991.

[261] Neil Postman. Informing ourselves to death. In *Computers, Ethics, and Society*. Oxford University Press, USA, third edition, November 2002. Reprinted from 1990.

[262] Miodrag Potkonjak and Jan Rabaey. Algorithm selection: a quantitative computation-intensive optimization approach. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*, pages 90–95, Los Alamitos, CA, USA, 1994. IEEE Computer Society.

[263] Roldan Pozo and Bruce Miller. http://math.nist.gov/scimark2/. National Institute of Standards and Technology (NIST), Accessed 7/2010.

[264] C. Priami. Stochastic pi-calculus. *The Computer Journal*, 38(7):578–589, July 1995.

[265] R. Prodan, T. Fahringer, and F. Franz. On using ZENTURIO for performance and parameter studies on cluster and Grid architectures. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 185–192, 2003.

[266] P. Pudlak. Complexity theory and genetics: The computational power of crossing over. *Information and Computation*, 171(2):201–223, December 2001.

[267] Francesco Quaglia. A middleware level active replication manager for high performance HLA-based simulations on SMP systems. In *Proceedings of the 10th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '06)*, pages 219–226, Washington, DC, USA, 2006. IEEE Computer Society.

[268] Francesco Quaglia. Software diversity-based active replication as an approach for enhancing the performance of advanced simulation systems. *International Journal of Foundations of Computer Science*, 18(3):495–515, 2007.

[269] J. R. Quinlan. Learning with continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992.

[270] Naren Ramakrishnan and Raúl Valdés-Pérez. Note on generalization in experimental algorithmics. *ACM Transactions on Mathematical Software*, 26(4):568–580, 2000.

[271] George Reese, Randy J. Yarger, Tim King, and Hugh E. Williams. *Managing and Using MySQL*. O'Reilly Media, second edition, April 2002.

[272] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[273] Jussi Rintanen. Phase transitions in classical planning: An experimental study. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 101–110. AAAI, 2004.

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

[275] Raul Rojas. *Neural Networks: A Systematic Introduction*. Springer, first edition, July 1996.

[276] Robert Rosen. *Life itself: a comprehensive inquiry into the nature, origin, and fabrication of life*. Columbia University Press, 1991.

[277] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, Inc., fourth edition, 1985.

[278] A. L. Ruhkin. Testing randomness: A suite of statistical procedures. *Theory of Probability and its Applications*, 45(1):111–132, 2001.

[279] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, December 2002.

[280] Stefan Rybacki, Jan Himmelspach, and Adelinde M. Uhrmacher. Experiments with single core, multi core, and GPU based computation of cellular automata. In *Proceedings of the SIMUL 2009*, pages 62–67. IEEE Computer Society, September 2009.

[281] Werner Sandmann. Simultaneous stochastic simulation of multiple perturbations in biological network models. In *Computational Methods in Systems Biology (CMSB 2007)*, pages 15–31. Springer Berlin / Heidelberg, 2007.

[282] Howard Sankey. Scientific method. In Stathis Psillos and Martin Curd, editors, *The Routledge Companion to Philosophy of Science*, Routledge Philosophy Companions, chapter 9, pages 248–258. Taylor & Francis, January 2008.

[283] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85, April 2009.

[284] Eric D. Schneider and James J. Kay. Life as a manifestation of the second law of thermodynamics. *Mathematical and Computer Modelling*, 19(6-8):25–48, 1994.

[285] B. Segal, L. Robertson, F. Gagliardi, and F. Carminati. Grid computing: the european data grid project. In *47th IEEE Nuclear Science Symposium and Medical Imaging Conference*, October 2000.

[286] William F. Sharpe. Mutual fund performance. *The Journal of Business*, 39(1):119–138, 1966.

[287] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J. P. Grossman, Richard C. Ho, Douglas J. Lerardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine Mcleavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, and Stanley C. Wang. Anton, a special-purpose machine for molecular dynamics simulation. *Communications of the ACM*, 51(7):91–97, July 2008.

[288] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, New York, NY, USA, December 2002. ACM Press.

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

[290] Christopher Small, Narendra Ghosh, Hany Saleeb, Margo Seltzer, and Keith Smith. Does systems research measure up? Technical Report TR-16-97, Harvard University, November 1997.

[291] J. S. Smith, J. A. Hamilton, R. E. Nance, B. L. Nelson, G. F. Riley, and L. W. Schruben. Panel discussion: What makes good research in modeling and simulation: Assessing the quality, success, and utility of M&S research. In *Proceedings of the 40th Winter Simulation Conference*, pages 689–694, 2008.

[292] Kate A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, 41(1):1–25, 2008.

[293] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/benchmarks.html. Accessed 7/2010.

[294] Ashok Srinivasan, Michael Mascagni, and David Ceperley. Testing parallel random number generators. *Parallel Computing*, 29(1):69–94, January 2003.

[295] R. Srivastava, L. You, J. Summers, and J. Yin. Stochastic vs. deterministic modeling of intracellular viral kinetics. *Journal of Theoretical Biology*, 218(3):309–321, October 2002.

[296] Karl-Georg Steffens. *The History of Approximation Theory: From Euler to Bernstein*. Birkhäuser, first edition, January 2006.

[297] David Strippgen and Kai Nagel. Using common graphics hardware for multi-agent traffic simulation with CUDA. In *Procceedings of the SIMUTools 2009*, 2009.

[298] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. MIT Press, May 1998.

[299] Kouichi Takahashi, Kazunari Kaizu, Bin Hu, and Masaru Tomita. A multi-algorithm, multi-timescale method for cell simulation. *Bioinformatics*, 20(4), 2004.

[300] Nassim N. Taleb. *The Black Swan. The Impact of the Highly Improbable*. Random House Inc., March 2008.

[301] Valerie Taylor, Xingfu Wu, and Rick Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):13–18, March 2003.

[302] Peihan Teo, Stephen J. Turner, and Zoltan Juhasz. Optimistic Protocol Analysis in a Performance Analyzer and Prediction Tool. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS '05)*, pages 49–58, 2005. http://dx.doi.org/10.1109/PADS.2005.17.

[303] University of Rostock The modeling & simulation research group. http://jamesii.org. Accessed 7/2010.

[304] Georgios Theodoropoulos, Rob Minson, Roland Ewald, and Michael Lees. Simulation engines for multi-agent systems. In Danny Weyns and Adelinde M. Uhrmacher, editors, *Agents, Simulation and Applications*, Computational analysis, synthesis, and design of dynamic models, chapter 3, pages 77–105. Taylor and Francis, 2009.

[305] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[306] Dominic P. Tolle and Nicolas Le Novère. Particle-based stochastic simulation in systems biology. *Current Bioinformatics*, 1(3):1–6, 2006.

[307] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1937.

[308] Adelinde Uhrmacher, Jan Himmelspach, Matthias Jeschke, Mathias John, Stefan Leye, Carsten Maus, Mathias Röhl, and Roland Ewald. One modelling formalism & simulator is not enough! A perspective for computational biology based on James II. In *Proceedings of the 1st International Workshop on Formal Methods in Systems Biology*, volume 5054 of *Lecture Notes in Bioinformatics*, pages 123–138. Springer Berlin / Heidelberg, 2008.

[309] Adelinde M. Uhrmacher, Roland Ewald, Mathias John, Carsten Maus, Matthias Jeschke, and Susanne Biermann. Combining micro and macro-modeling in DEVS for computational biology. In *Proceedings of the 39th Winter Simulation Conference*, pages 871–880, 2007.

[310] Adelinde M. Uhrmacher, Jan Himmelspach, Mathias Röhl, and Roland Ewald. Introducing Variable Ports and Multi-Couplings for Cell Biological Modeling in DEVS. In *Proceedings of the 38th Winter Simulation Conference*, 2006.

[311] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2000. IEEE Computer Society.

[312] N. Vaidehi and W. A. Goddard. Atomic-level simulation and modeling of biomacromolecules. In James M. Bower and Hamid Bolouri, editors, *Computational modeling of genetic and biochemical networks*, pages 161–188. MIT Press, 2001.

[313] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[314] Wilfred F. van Gunsteren and Herman J. Berendsen. Computer simulation of molecular dynamics: Methodology, applications, and perspectives in chemistry. *Angewandte Chemie International Edition in English*, 29(9):992–1023, 1990.

[315] J. van Leeuwen and J. Wiedermann. The turing machine paradigm in contemporary computing. Technical Report UU-CS-2000-33, Department of Information and Computing Sciences, Utrecht University, 2000.

[316] Joannès Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Machine Learning: ECML 2005*, volume 3720/2005 of *Lecture Notes in Computer Science*, pages 437–448. Springer Berlin / Heidelberg, 2005.

[317] Richard Vuduc, Jeff Bilmes, and James Demmel. Statistical modeling of feedback data in an automatic tuning system. In *MICRO-33: Third ACM Workshop on Feedback-Directed Dynamic Optimization*, 2000.

[318] Richard Vuduc, James Demmel, and Jeff Bilmes. Statistical models for automatic performance tuning. In *Computational Science – ICCS 2001*, volume 2073/2001 of *Lecture Notes in Computer Science*, pages 117–126. Springer Berlin / Heidelberg, 2001.

[319] Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. Statistical models for empirical search-based performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, February 2004.

[320] David Waltz and Bruce G. Buchanan. Computer science: Automating science. *Science*, 324(5923):43–44, April 2009.

[321] Bing Wang, Jan Himmelspach, Roland Ewald, Yiping Yao, and Adelinde M. Uhrmacher. Experimental analysis of logical process simulation algorithms in JAMES II. In M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, editors, *Proceedings of the 41st Winter Simulation Conference*, pages 1167–1179. IEEE Computer Science, 2009.

[322] Jun Wang and Carl Tropper. Optimizing time warp simulation with reinforcement learning techniques. In *Proceedings of the 39th Winter Simulation Conference*, pages 577–584, Piscataway, NJ, USA, 2007. IEEE Press.

[323] Jun Wang and Carl Tropper. Using genetic algorithms to limit the optimism in time warp. In M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, editors, *Proceedings of the 41st Winter Simulation Conference*, pages 1180–1188. IEEE Computer Science, December 2009.

[324] S. Weerawarana, E. N. Houstis, J. R. Rice, A. Joshi, and C. E. Houstis. PYTHIA: a knowledge-based system to select scientific algorithms. *ACM Transactions on Mathematical Software*, 22:447–468, 1996.

[325] Karsten Weihe. On the differences between "practical" and "applied". In *Algorithm Engineering: 4th International Workshop, WAE 2000*, volume 1982/2001 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin / Heidelberg, 2001.

[326] Joseph Weizenbaum. *Computer Power and Human Reason: From Judgement to Calculation.* W.H. Freeman & Company, March 1976.

[327] Weka 3: Data Mining Software in Java. http://www.cs.waikato.ac.nz/ml/weka/. Accessed 7/2010.

[328] D. Weyns, T. Holvoet, and A. Helleboogh. Anticipatory vehicle routing using delegate multi-agent systems. In *Intelligent Transportation Systems Conference (ITSC 2007)*, pages 87–93. IEEE, 2007.

[329] Clint R. Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[330] Douglas J. White. Real applications of Markov decision processes. *Interfaces*, 15(6):73–83, 1985.

[331] T. Wilding. Using genetic algorithms to construct portfolios. In Stephen Satchell and Alan Scowcroft, editors, *Advances in Portfolio Construction and Implementation*, Quantitative Finance Series, chapter 6, pages 135–160. Butterworth-Heinemann, June 2003.

[332] James Wilson. Responsible authorship and peer review. *Science and Engineering Ethics*, 8(2):155–174, June 2002.

[333] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, October 1999.

[334] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[335] William A. Wulf. A case against the GOTO. In *ACM '72: Proceedings of the ACM Annual Conference*, pages 791–797, New York, NY, USA, 1972. ACM.

[336] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.

[337] Qing Xu and Carl Tropper. On determining how many computers to use in parallel VLSI simulation. In *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*, pages 122–128. IEEE CPS, June 2009.

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

[339] L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-3(1):28–44, January 1973.

[340] Bernard P. Zeigler, Herbert Praehofer, and Tag G. Kim. *Theory of Modeling and Simulation.* Academic Press, second edition, 2000.

[341] Ming Zhang, Bernard P. Zeigler, and Phillip Hammonds. DEVS/RMI – an auto-adaptive and reconfigurable distributed simulation environment for engineering studies. *ITEA Journal Of Test And Evaluation*, 27(1), 2006.

[342] Yuan Zhao, Qing Yi, Ken Kennedy, Dan Quinlan, and Richard Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical Report UCRL-TR-217808, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, December 2005.

List of Figures

List of Tables

Listings

Index