Toward Guiding Simulation Experiments

Dissertation

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

vorgelegt von:

Stefan Leye geb. am 16.12.1982 in Neubrandenburg

Rostock, 25. Oktober 2013

urn:nbn:de:gbv:28-diss2014-0117-7

Gutachter:

- Prof. Dr. Adelinde Uhrmacher, Institut für Informatik, Universtät Rostock
- Prof. Luiz Felipe Perrone, Dept. of Computer Science, Bucknell University
- Prof. Corrado Priami, Centre for Computational and Systems Biology, University of Trento

Verteidigt am 7. Januar 2014

Abstract

The plethora of methods available for simulation experiments can be overwhelming for users. Different tasks have to be considered, and selecting the most suitable method for the task at hand is challenging. As a consequence, tools are required that allow the seamless integration of various methods, guide users through the tasks of the experiment, and support them in deciding which methods to use.

This work presents techniques for facing such challenges, which have been developed as part of the environment for *GUIding Simulation Experiments (GUISE)*. To enable guidance, typical tasks to be considered in a simulation experiment are identified in a review on existing literature and tools on the matter. As a result, six tasks are derived, i.e., specification of the experiment, configuration of model input parameters, model execution, collection of simulation output data, data analysis, and result evaluation.

The experimentation layer of the modeling and simulation framework JAMES II is extended to provide explicit support for such tasks, by exploiting JAMES II's plugin system. Example experiments show the applicability of the approach in four case studies, including a validation experiment in molecular biology, a sensitivity analysis and an optimization experiment in cellular biology, as well as an exploratory analysis experiment in electrical engineering. In addition, the experiment structure is mapped to workflows by using the workflow management system WorMS which ships with documentation and data provenance features that help making experiments reproducible.

To assist users in finding the most suitable solution for facing a task, a synthetic problem solver is introduced which composes existing algorithms for the problem at hand. The composition is done by considering problem and algorithm features, and complies with various selection & composition schemes like algorithm selection, portfolios, or ensembles. Synthetic problem solvers are made accessible in an extension of the simulation algorithm selection framework SASF and can be easily deployed as JAMES II plugins. Their effectiveness is illustrated in two case studies dealing with algorithms for steady state mean estimation and statistical analyzers for estimating the number of required replication.

CCS–Classification (2012)::
- Computing methodologies∼Simulation tools
- Computing methodologies∼Simulation evaluation
- Information systems∼Decision support systems
- Social and professional topics∼Software selection and adaptation
- Computing methodologies∼Ensemble methods
- Computing methodologies∼Feature selection

Keywords:
modeling and simulation, simulation experiments, simulation output analysis, scientific workflows, adaptive software, algorithm selection problem, algorithm portfolios, algorithm ensembles

Zusammenfassung

Die Menge an verfügbaren Methoden, die in Simulationsexperimenten einsetzbar sind, kann Nutzer überfordern. Eine besonderes Problem ist die Auswahl der richtigen Methoden für die verschiedenen Aufgaben, die ein Simulationsexperiment beinhaltet. Daher sind Werkzeuge notwendig, die das unproblematische Einbetten der Methoden, eine Führung durch die einzelnen Aufgaben, sowie Unterstützung bei der Auswahl von Methoden für diese Aufgaben bieten.

Die vorliegende Arbeit behandelt Techniken, die für die Umgebung zur Nutzerunterstützung von Simulationsexperimenten GUISE entwickelt wurden, um diesen Herausforderungen zu begegnen. Es werden Literatur und Werkzeuge untersucht um typische Aufgaben, die in einem Simulationsexperiment auftreten, zu identifizieren. Dabei werden sechs Aufgaben abgeleitet: die Spezifikation des Experiments, die Konfiguration von Modelparametern, die Ausführung von Modellen, die Sammlung von Simulationsdaten, deren Analyse sowie die Auswertung der Ergebnisse. Daraufhin wird die Experimentierschicht des Simulationsrahmenwerkes JAMES II erweitert um diese Aufgaben, unter zu Hilfenahme des Plugin-Systems von JAMES II, explizit zu unterstützen. Vier Beispielexperimente untermauern die Anwendbarkeit des Konzeptes. Dies beeinhaltet ein Validierungsexperiment in der Molekularbiologie, eine Sensitivitätsanalyse sowie ein Optimierungsexperiment in der Zellbiologie und ein exploratives Experiment in der Elektrotechnik. Außerdem wird das Konzept auf Arbeitsabläufe abgebildet, um die Dokumentationseigenschaften des Workflow-Management-Systems WorMS auszunutzen, die insbesondere helfen Experimente reproduzierbar zu machen.

Um Nutzern bei der Auswahl der passenden Methode für eine Aufgabe zu helfen, wird ein Konzept zur synthetischen Problemlösung eingeführt, das Algorithmen komponiert um gegebene Probleme zu lösen. Die Komposition betrachtet Problem und Algorithmeneigenschaften und erlaubt die Abbildung verschiedener Kompositionsstrategien, wie Algorithmenselektion, Portfolios oder Ensembles. Die synthetische Problemlösung wird in eine Erweiterung des Rahmenwerkes zur Simulationsalgorithmenselektion SASF eingebettet und konkrete Kompositionen können als JAMES II Plugins eingesetzt werden. Ihre Effektivität wird in zwei Fallstudien gezeigt, die einerseits Algorithmen zur Abschätzung von Steady-State-Statistiken und andererseits Analysewerkzeuge zur Abschätzung von notwendigen Replikationen behandeln.

CCS–Classification (2012)::
• Computing methodologies∼Simulation tools
• Computing methodologies∼Simulation evaluation
• Information systems∼Decision support systems
• Social and professional topics∼Software selection and adaptation
• Computing methodologies∼Ensemble methods
• Computing methodologies∼Feature selection

Keywords:
Modellbildung und Simulation, Simulationsexperimente, Analyse von Simulationsdaten, wissenschaftliche Arbeistabläufe, adaptive Software, Algorithmenauswahl,Algorithmenportfolios, Algorithmenensembles

Acknowledgments

During the development of this thesis, I benefited from the healthy and productive environment my supervisor Lin Uhrmacher created in the modeling and simulation group at the University of Rostock. I consider myself very lucky to be part of it, and think that she found the perfect balance between self-dependent work and constructive advice for her PhD students. The outstanding patience and support Lin gave me over the years were essential for my progress, and I am very grateful for that.

Furthermore, I am very thankful to my present and former colleagues from the working group, in particular, Roland Ewald and Jan Himmelspach. Roland did not only support me in developing the synthetic problem solver and extending SASF, but he was always helpful and dedicated anytime I came to him with a problem. He is one of the kindest guys I have ever known, and I enjoyed our discussions about anything and everything during our (more or less) weekly running sessions.

Jan accompanied the better part of my scientific activity so far. He did not only advise my student's and diploma theses, but gave me useful hints and support after my graduation. I cannot say that I took his constructive criticisms always easy but they were every time inspiring and thought-provoking, and I consider Jan as one of my academic mentors.

Stefan Rybacki supported me in using WorMS and provided fruitful suggestion for developing the experiment workflow. Matthias John, Fiete Haack, Orianne Mazemondet, and Tim Wegner developed the models and simulation engines that were subject of the example experiments used for evaluating the experiment concepts developed in this work. James Faeder suggested the TCR model for evaluating steady state estimators and statistical analyzers, and Tobias Helms implemented the ML-Rules version of this model. Felipe Perrone did not only review this thesis, but provided very helpful comments for the final version.

To be productive, the time for rest and regeneration is similarly important as the working time. Therefore, I thank Alexander Steiniger for accompanying me during our lunch breaks, and the stimulating discussions we had on the way to the Asian diner.

Special thanks go to my girlfriend, who helped me, in her inspiring and loving manner, to escape from my workdays and recharge my batteries. Last but not least, I am deeply grateful for my parents, who, like no one else, supported me throughout my whole life in the things I did, always giving me the feeling that they are proud of me.

Contents

1. Introduction

"I've always felt that the human-centered approach to computer science leads to more interesting, more exotic, more wild, and more heroic adventures than the machine-supremacy approach, where information is the highest goal."

Jaron Lanier

1.1. Motivation

What is the reason of the experiment? This is a question every empirical scientist should have heard or asked himself during his career. When Benjamin Franklin proposed his famous experiment of collecting electricity with a conductive kite [56], the answer to this question would have been: to prove the electrical nature of lightning. Many experiments have been, and are, executed for very good reason. The Large Hadron Collider (LHC) [47] at the European Organization for Nuclear Research (CERN), for instance, shall investigate not less than the inner workings of the universe.

However, despite the good questions they pursue, the settings of some experiments, like holding a kite into the sky waiting for a lightning to struck it, or smashing together tiny subatomic particles in a pipe of 26 kilometers length, are puzzling at first glance. Nevertheless, both experiments follow a reasonable design resulting from scientific considerations. With the rising complexity of information we gain about our world, the experiments, conducted to get (or validate) new information, rise in their complexity as well. As a result, many research fields put effort into the design and execution of experiments [170, 173, 99, 116].

In the past, these fields usually dealt with supporting *traditional experiments*, where the system under investigation is part of reality, e.g., a thundercloud and a kite in Franklin's experiment, or subatomic particles in the LHC. Handling such systems, however, can be challenging. In many cases real systems are too big, too small, too expensive, or too complex to be controlled properly. The LHC, for instance, deals with the smallest known particles in one of the largest facilities of the world, constructed by employing state of the art engineering techniques. In addition to these technical and economic issues, executing an experiment is also questionable if lives are at stake. For instance, it is not desirable to test the protection system of a car with a real human inside.

A different kind of experiment can be used to circumvent such issues. *Simulation experiments* do not focus on the system itself, but on an abstraction of it. This abstraction, or *model*, does not have all the properties of the underlying system, making it easier to handle. As an additional benefit, the system is not affected by the experimentation process.

The model can be, e.g., a different item of reality (like a crash test dummy) or a mathematical representation. The latter gained more and more importance with the invention and success of computers, as the field of *Modeling and Simulation (M&S)* was established, making mathematical models more easily executable and analyzable. This easier execution and control of models compared to real systems, made simulation an important tool of scientific work. Due to its synthetic nature, observing a model is often more easy than observing the real system. It is for instance easily possible, to keep track of the population density in a demographic model, while collecting these data in reality, is much more difficult [11]. With the ease of collecting data, unfortunately, the problem of handling them is not solved. Experimenters still have to choose which data to collect and need to process raw data to get meaningful experiment outcomes.

A similar problem arises with the option to control a model. In contrast to real systems, virtually any aspect of a model can be controlled. A real chemical solution for instance has to be treated by rather indirect methods, e.g., injecting other chemical substances, whereas a chemical reaction network can be directly altered by changing reaction rates or even altering the reaction rules. Thus, many potential control inputs have to be considered. For both tasks, input control and data processing, various methods exist [10] that are subject of scientific research, e.g., optimization [153], or steady state estimation [8]. Additional tasks (e.g., model execution) exist, which are not less challenging. As a result, special care has to be put into the setup of a simulation experiment, as it involves a variety of intertwined tasks.

Literature on simulation experiments demands a better support, as users often lack the required mathematical background for a systematic application of methods [86, 89, 147]. As Perrone et. al. [147] state, "the level of complexity of rigorous simulation methodology requires more from networking researchers than they are capable of handling without additional support from software tools." This statement is likely to be applicable to any field where simulation experiments play a role. Recent publications even identified a "crisis of credibility" in simulation studies [146, 118].

One approach to tackle this problem is structuring the experiment process and identifying specific steps that need recognition and support equally, like done in [121] for validation experiments. A tool is required which implements this structure and allows a seamless integration of methods, making them available to users. To achieve this, a scientific workflow [14, 97, 3, 160] can be beneficial. Integrated as part of a simulation tool, a workflow can guide the user and create documentation for the simulation experiment.

However, the challenges do not end with structuring the experiment process. While it is beneficial to know, which tasks have to be executed at which point in the simulation study, it is not clear how these tasks are fulfilled. Usually, many different methods are applicable for one task or problem (like steady state estimation [45, 193, 166, 144, 27, 190, 120, 111]), and the suitability of each method depends on the problem's features.

The diversity of methods has been recognized by developers of simulation tools, who put effort into a generalization of the experiment process to integrate different methods for experiment tasks. Examples for resulting tools are AnyLogic [31] which has been designed to combine methods for different experiment tasks, like simulation, analysis, and visualization, or BioCham [52] with its ability to investigate temporal properties of biological models in combination with parameter search and scan strategies. The trend for generalization culminates in multi-purpose M&S frameworks like JAMES II [85, 83] which faces the various methods and aspects of modeling and simulation by providing a flexible and extensible tool, that allows to integrate and reuse methods.

While this poses a first step in helping non-domain experts in managing the different available methods — as they have to put less effort into combining them — a support in selecting the right method is missing. Such support is required, as Wolpert and Macready [195] show with their *no free lunch* theorem which states that no optimization algorithm exist, that finds optimal solutions regardless from the given problem. Asmussen [8] formulates a similar statement for steady state estimation and proved that no universal solution exists for finding the end of the warm-up phase of a time-series, i.e., no algorithm for that purpose *always* produces correct results (independently of specific time-series features). However, it is still possible to construct a solution that performs particularly well for *certain* kinds of problems, e.g., those that are particularly relevant in practice.

Hence, guidance is required to help users in selecting proper methods for various problems. To support this, general-purpose simulation systems need to offer generic composition mechanisms, which should be easy to tune to the given application domain for improving performance. This is, for instance, relevant in computational systems biology, as Ghosh et. al. [64] demand that *"software tools and resources for systems biology need to be tailored to their intended applications in order to achieve the objectives of novel biological discoveries"*.

1.2. Terminology

The field of M&S focuses on the three fundamental terms: *model*, *simulation*, and (attributed to Minsky [137]) *experiment*.

Cellier states that *"a model for a system S and an experiment E is anything to which E can be applied in order to answer questions about S"* [29, p. 4]. According to this definition, a model is always linked to a system (which is usually but not necessarily part of reality, e.g., a biological cell, a demographic population, or the weather system).

Furthermore, the model shall answer questions about the system, i.e., it is always built with a specific purpose or question, like: how will the cell proliferate; will the population survive over the next 10 years; or how will the weather change during the next days. Usually, such questions cannot easily be answered by performing experiments on the system itself, because those experiments might be too expensive, complex, take too much time, or the system being too small to be observed, etc.

Finally, a model is an abstraction of the modeled system that contains, at least, those features of the reference systems that are required to answer the raised questions.

To put it in a nutshell, a model has the following three key properties:

- Reference: the model has to represent a given system.
- Abstraction: the model comprises a selected subset of the system's features.
- Purpose: the model shall answer one or more questions.

All three properties are closely related: a model shall comprise those features of the reference system that are required to answer the given questions.

The second fundamental term of M&S is *simulation*. Bratley defines simulation as the process of *"driving the model with certain [...] inputs and observing the corresponding outputs"* [24, p. 2]. This definition characterizes simulation as a single model execution (with given input).

Korn, on the contrary, has a shifted viewpoint on the term, when stating that *"a simulation is an experiment performed on a model"* [117, p. 8]. He stresses the experimental nature of simulation. An experiment can be a very complex and multi-faceted task that has been the focus of broad scientific research [199, 10, 162, 83]. Cellier describes experiments as being "the process of extracting data from a system by exerting it through its inputs" [29, p. 5]. The experiment considers the underlying system, as a black box. It is not necessary to define what a system is exactly, as long as it provides the functionality of taking inputs and producing outputs. Consequently, just the relation between input and output is of interest during an experiment, whereas the internals of the system are not relevant. Generating interesting input parameters and producing substantial output data is essential for conducting fruitful experiments.

The two opinions on the term simulation — as single model execution or as an experiment on a model — shall be distinguished in the following by using the terms *simulation run* for the first, and *simulation experiment* for the latter. The verb *to simulate* will refer to executing a model.

In stochastic simulation it is desirable to repeat a simulation run with different random seeds but the same simulation and model parameters. This is reflected by the term *replication* which will be used synonymously to (a repeated) simulation run.

In addition to the previously defined terms, *simulation study* refers to a set of simulation experiments that answer a hypothesis. For instance, the hypothesis that a cell proliferates with a given rate could be answered in a simulation study working on a model of the cell and comprising a validation experiment for showing that the model behaves similar as the cell, as well as an exploratory experiment for testing the hypothesis.

In the following some additional terms of experimentation in the context of M&S, relevant for this work, will be explained.

Algorithm/Problem Solver In general, computation is done by *algorithms*. Turing proposed a well-accepted formalization for algorithms with his *Turing Machine* [175, 176] that characterizes how the term is understood in this work.

Usually, algorithms work on some problem and produce a result or solution. The term *problem solver* stresses this behavior, and both terms, algorithm and problem solver, will be used synonymously in the following.

Simulator/Simulation Engine/Simulation Algorithm Usually, a model is implemented by using a modeling language. By applying the semantics of that language the model can be simulated, i.e., executed. Algorithms that realize these semantics (meaning they are able to execute a model in the given language) are called *simulators*. The terms *simulation engine* and *simulation algorithm* will be used synonymously to simulator.

Parameters, Configurations, and Corresponding Methods The input of a model depends on its parameters which here also include initialization values, like the initial numbers of species or rates in chemical reaction networks. In order to execute a model, a value has to be determined for each parameter. A combination of values for the parameters of a model will be called *input parameter setting* or simply *parameter setting*. A model instantiated with a set of input values will be called *model configuration* or simply *configuration*.

Note that the term configuration also appears with respect to the input parameters of a simulation, e.g., used simulation engine, random number generator, or stop condition. In that case the term *simulation configuration* is used.

Methods for searching the parameter space for a setting that fulfills certain conditions (e.g., a minimal objective) will be called *parameter search methods*. In contrast, methods that investigate the behavior of a given set of parameter settings will be called *parameter scan methods*.

Configuration Run Usually, it is necessary to investigate different configurations during a simulation experiment. In the stochastic case, each configuration has to be executed multiple times, i.e., multiple replications have to be performed. The term *configuration run* will be used for the set of replications that execute the same configuration.

Tasks and Workflows Scientific problem solving comprises the execution of many interdependent tasks. Scientific workflows are used to describe, document, and synchronize those tasks in order to support and structure them.

According to Van der Aalst the term *workflow* is defined as *"a network of tasks with rules that determine the (partial) order in which the tasks should be performed. A rule can be either "Sequencing", "Selection of choice", or "Parallelism"* [181, p. 25]. Hence, workflows are based on tasks that can be composed sequentially (i.e., one task is executed after another), in parallel, or by selection (a decision has to be made between different tasks and only one is executed).

Tasks are defined as *"a logical unit of work [that] is indivisible and thus always carried out in full"* [181, p. 32]. A task is the most basic unit of a workflow and cannot be interrupted by other tasks. However, it can be further structured in sub-workflows that are self-contained. In the M&S field, a task could be, for instance, the execution of a simulation step.

1.3. Contribution

This work aims at the conception and implementation of user support strategies for conducting simulation experiments. Such strategies are combined in the *environment for GUIding Simulation Experiments (GUISE)* [126]. As the abbreviation implies, GUISE focuses on simulation experiments. Other experiment types surrounding M&S (e.g., performance experiments with simulation engines) are not specifically covered, even though using GUISE for such purposes is not precluded.

The user support strategies introduced in this work focus on the following two questions:

1. Which tasks have to be considered during a simulation experiment?
2. Which algorithms shall be used to execute a task?

To cover the first question literature and tools from the M&S realm are reviewed, and six tasks, that are typically part of a simulation experiment, identified. The task are the specification of the experiment, configuration of input parameter settings, model execution, data collection, data analysis, and result evaluation. For supporting such tasks explicitly, the experimentation layer of the plugin based M&S framework JAMES II, designed by Himmelspach et. al. [85], is extended. In particular, the tasks specification, configuration, analysis, and evaluation are realized by implementing corresponding plugin types. Proof of concept is given in four different scientific studies that applied the extended experimentation layer. This includes a validation experiment within molecular biology, a sensitivity analysis and an optimization experiment in cellular biology, as well as an exploratory analysis experiment in electrical engineering,

The identified experiment structure is mapped to a workflow, by exploiting the workflow management system WorMS [160] and its notion of templates and frames. The workflow benefits from WorMS' documentation features, which helps making experiments reproducible and contributes to the credibility of simulation studies.

To answer the second question, i.e., deciding which algorithms shall be used to execute a task, automatic support for configuring simulation experiments are integrated into the experiment structure. Therefore, a synthetic problem solver is designed, that composes algorithms according to algorithm and problem features to find the most suitable solution in the given situation. The creation and usage of synthetic problem solvers is prototyped by extending the *Simulation Algorithm Selection Framework (SASF)*, developed by Ewald [48]. The generality of the approach is shown in a discussion of how existing algorithm selection & composition techniques can be realized as synthetic problem solvers. The applicability of the synthetic problem solver is shown in two case studies concerning steady state estimation and algorithms for the statistical estimation of replication numbers.

1.4. Outline

This thesis comprises two main parts. The first part deals with structuring the experiment process and implementing the resulting structure. Chapter 2 reviews existing concepts and tools for simulation experiments and derives the tasks to be considered when conducting them. To integrate these tasks, Chapter 3 extends the experimentation layer of the simulation framework JAMES II and illustrates the flexibility and applicability of the extension in example experiments. Chapter 4 presents a workflow mapping of the experiment structure, based on the workflow management system WorMS.

The second part of the thesis deals with the integration of selection & composition mechanisms into the experiment structure. Chapter 5 extends the algorithm selection framework SASF for algorithms used in simulation experiments, and introduces synthetic problem solvers that compose algorithms by applying selection & composition mechanisms. Chapters 6 and 7 present case studies dealing with the creation of synthetic problem solvers for steady state estimation and statistical analysis.

The work is finished with a conclusion and an outlook about future work.

1.5. Bibliographic Note

The six tasks of a simulation experiment, that are proposed in Chapter 2, have been first identified for validation experiments in the following publication which surveyed validation in M&S.

S. Leye, J. Himmelspach, and A. M. Uhrmacher. A discussion on experimental model validation. In *Proceedings of the International Conference on Computer Modelling and Simulation (UKSIM)*, pages 161–167, 2009

The experimentation layer extension, based on these tasks and discussed in Chapter 3, has been presented in the publication below. It argued that the extension is general and flexible enough to allow a seamless integration of arbitrary experimental validation methods. While the extension focused on validation experiments, later applications showed that it can be used for different experiment types, as well.

> S. Leye and A. M. Uhrmacher. A flexible and extensible architecture for experimental model validation. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools)*, pages 65:1–65:10, 2010

The first substantial experiment executed with the extended experimentation layer was a validation experiment of a NAM method reimplementation that has been developed by Fiete Haack. It has been published in the following conference article and provided valuable insight into different ways of executing a model for brownian dynamics.

> F. Haack, S. Leye, and A. M. Uhrmacher. A flexible architecture for modeling and simulation of diffusional association. In *Workshop From Biology To Concurrency and back (FBTC)*, pages 70–84, 2010

A second experiment, that has been conducted using the experimentation layer extension, has been described in the following journal article. It presented a Wnt/β-catenin model, developed by Orianne Mazemondet, and described the first application of the extended experimentation layer not dealing with a validation experiment, but with a sensitivity analysis of the model.

> O. Mazemondet, M. John, S. Leye, A. Rolfs, and A. M. Uhrmacher. Elucidating the sources of β-catenin dynamics in human neural progenitor cells. *Public Library of Science (PLoS) ONE*, 7(8):1–12, 2012

In addition to the general design of the experimentation layer, effort has been put into parallelizing its execution. The following conference article described the mechanics to parallelize the simulation output analysis of multiple replications.

> S. Leye, O. Mazemondet, and A. M. Uhrmacher. Parallel analysis with FAMVal to speed up simulation-based model checking. In *European Symposium on Computer Modeling and Simulation (EMS)*, pages 344–350, 2010

The workflow mapping of the experiment structure, which will be described in Chapter 4, is the result of joint work with Stefan Rybacki and has been presented in the following publication. It stressed the benefits of frames and templates that ship with WorMS, to realize a simulation experiment workflow.

> S. Rybacki, S. Leye, J. Himmelspach, and A. M. Uhrmacher. Template and frame based experiment workflows in modeling and simulation software with WORMS. In *Proceedings of the International Workshop on Scientific and Engineering Workflows: Advances in Data and Event-Driven Workflows (SWF)*, pages 25–32, 2012

An overall overview over the support strategies, discussed in this thesis, has been presented in the following conference poster. It introduced the selection & composition mechanism, i.e., the synthetic problem solver, and illustrated its interaction with the experiment workflow. Furthermore, first results of the steady state estimation case study (see Chapter 6) have been shown.

> S. Leye and A. M. Uhrmacher. GUISE - a tool for GUIding Simulation Experiments. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 305:1–305:2, 2012

The synthetic problem solver has been developed in joint work with Roland Ewald. A more detailed description of the concept and its application on steady state estimation has been submitted.

Part I.

Structuring a Simulation Experiment

2. Identifying a Common Structure for Simulation Experiments

"Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning."

Albert Einstein

Undoubtedly, the application of a simulation experiments increased with the invention and success of computers. To deal with the challenges and opportunities that resulted from the massive amounts of data generated by computers, effort has been put into organizing and structuring as well as developing algorithms for simulation experiments. Over the years, a plethora of approaches emerged in this field.

This chapter summarizes different publications and tools on the topic. The first section focuses on publications that deal with the organization and execution of simulation experiments, whereas the second section discusses simulation tools that have been developed in various application areas. Both sections conclude with a comparison of the presented approaches in order to identify relevant features to be considered during simulation experiments in general. The third section subsumes those features and derives a common structure for simulation experiments.

2.1. Experimentation in Academic Literature

Literature on simulation experiments will be reviewed in five parts. The first part (Section 2.1.1) deals with describing what an experiment is and how it is defined; the second part (Section 2.1.2) focuses on common questions and challenges that have to be dealt with when executing simulation experiments; the third part (Section 2.1.3) discusses processes that embed as well as processes that realize simulation experiments; the fourth part (Section 2.1.4) deals with goals that are pursued when conducting an experiment. Finally, the fifth part (Section 2.1.5) will summarize the gained insights.

The aim of this section is not to give a complete overview over existing literature on the topic, but to show which features are commonly considered in publications covering different viewpoints.

2.1.1. Basic Concepts

As stated in Section 1.2, p. 3, Cellier defines an experiment as *"the process of extracting data from a system by exerting it through its inputs"* [29, p. 4]. By considering models as systems, this definition is applicable to simulation experiments, and can be adjusted to: *a simulation experiment is the process of extracting data from a model by exerting it through its inputs.* This implies different questions that have to be regarded for executing simulation experiments (see Figure 2.1). Exerting the model through its inputs leads to the questions of how the system is exerted, and which inputs are of interest. The first question refers to executing the model, whereas the second refers to configuring its input parameters. Hence execution and configuration are two relevant tasks that have to be considered during a simulation experiment. Extracting data from the model leads to the questions of how to collect and analyze data, both directly referring to important tasks, i.e., data collection and analysis. Altogether, Cellier's definition addresses four experiment tasks: the configuration of model input parameters, model execution, data collection, and data analysis.

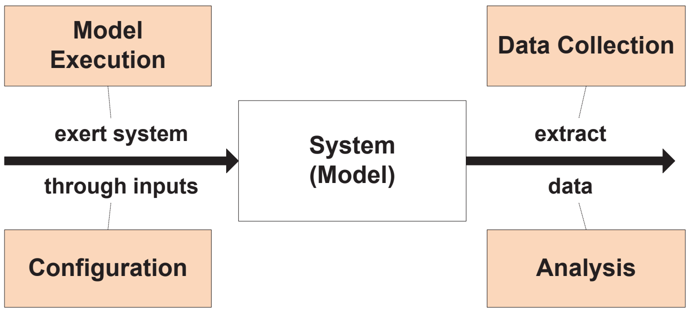


Figure 2.1.: Scheme for an experiment according to Cellier's definition [29, p. 4]. Orange boxes contain tasks that have to be considered.

Zeigler [199] introduced one of the first theoretical concepts realizing a distinct treatment of model and experiment that is essential for a separation of concerns in M&S. He proposes the *experimental frame for simulations*, as "a specification of the conditions under which the system is observed or experimented with" [199, p. 27]. It sets conditions for an experiment by determining, e.g., the mean rate of arrivals (for queueing systems), the seeds for random number generators, and model input parameters. According to Zeigler [143] the experimental frame should deal with five types of

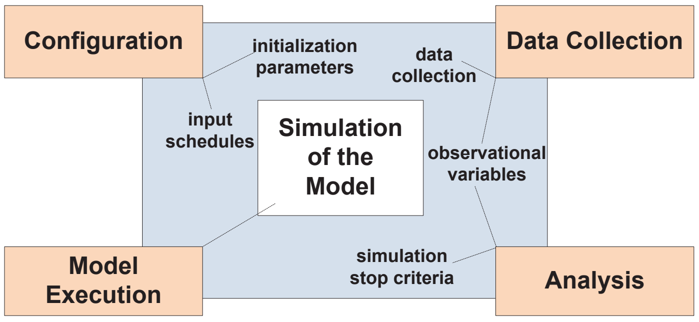


Figure 2.2.: The experimental frame according to [143]. The frame (blue box) is wrapped around the simulation, i.e., execution, of the model, and provides conditions for the execution. Orange boxes contain tasks which have to be performed to set such conditions.

conditions (illustrated in Figure 2.2): input schedules (trajectories), initialization parameters, observational variables, data collection, and simulation stop conditions. All of them refer to tasks that have to be considered during simulation experiments. Input schedules and initialization parameters are generated during the configuration of model input parameters. Observational variables are those variables that need to be directly observed, not merely inferred, and are an essential part of data collection and analysis. Simulation stop conditions, i.e., conditions determining when a simulation run is stopped, are based on analysis results. Altogether, the experimental frame demands three tasks to be considered, in addition to the central model execution task. These additional tasks are the configuration of model input parameters, data collection, and data analysis.

2.1.2. Experiment Overviews

Law and Kelton [10] describe several challenges of simulation experiments, including the selection of simulation software, experimental design and input distributions, generation of random numbers, analysis of simulation output, and the creation of reliable results. All of theses challenges refer to experiment tasks. The selection of simulation software addresses model execution, while experimental design, creation of input parameters, and generation of random numbers are required for configuring model input parameters properly. Analysis of outputs and creation of reliable results are part of the data analysis task.

In a different publication on simulation experiments, Kelton proposes a list of questions, an experimenter should consider [109, p. 32]:

- *"What model configurations should you run?"*
- *"How long should the runs be?"*
- *"How many runs should you make?"*
- *"How should you interpret and analyze the output?"*
- *"What's the most efficient way to make the run?"*

The first question deals with the configuration of input parameters, while the following three questions tackle the analysis of simulation output data, since length and amount of simulation runs have to be sufficient to get enough data for analysis. The last question is focused on simulating the model fast but accurately. Thus, by raising these questions, Kelton demands to deal with the experiment tasks: model parameter configuration, model execution, and data analysis, which are the same tasks that he and Law illuminate in [10].

The *Minimum Information About a Simulation Experiment (MIASE)* [186] is a standard proposing the minimal information that have to be provided in order to describe a simulation experiment. It has been specifically designed to make experiments repeatable. The MIASE guidelines are given as follows [186, p. 3]:

"1. All models used in the experiment must be identified, accessible, and fully described.

(a) The description of the simulation experiment must be provided together with the models necessary for the experiment, or with a precise and unambiguous way of accessing those models.

(b) The models required for the simulations must be provided with all governing equations, parameter values, and necessary conditions (initial state and/or boundary conditions).

(c) If a model is not encoded in a standard format, then the model code must be made available to the user. If a model is not encoded in an open format or code, its full description must be provided, sufficient to re-implement it.

(d) Any modification of a model (pre-processing) required before the execution of a step of the simulation experiment must be described.

2. A precise description of the simulation steps and other procedures used by the experiment must be provided.

(a) All simulation steps must be clearly described, including the simulation algorithms to be used, the models on which to apply each simulation, the order of the simulation steps, and the data processing to be done between the simulation steps.

(b) All information needed for the correct implementation of the necessary simulation steps must be included through precise descriptions or references to unambiguous information sources.

(c) If a simulation step is performed using a computer program for which source code is not available, all information needed to reproduce the simulation, and not just repeat it, must be provided, including the algorithms used by the original software and any information necessary to implement them, such as the discretization and integration methods.

(d) If it is known that a simulation step will produce different results when performed in a different simulation environment or on a different computational platform, an explanation must be given of how the model has to be run with the specified environment/platform in order to achieve the purpose of the experiment.

3. All information necessary to obtain the desired numerical results must be provided.

(a) All post-processing steps applied on the raw numerical results of simulation steps in order to generate the final results have to be described in detail. That includes the identification of data to process, the order in which changes were applied, and also the nature of changes.

(b) If the expected insights depend on the relation between different results, such as a plot of one against another, the results to be compared have to be specified."

The first point deals with models, and focuses mainly onto their representation, i.e., they should be made available and accessible, which is an important problem for repeatability. Subitem *1.(b)*, furthermore, stresses that the model shall be given with its parameter values (i.e., its input parameters), which directly leads to the question of how to configure parameter settings. The second point focuses on model execution, which should be described precisely, including used algorithms, their implementation, and environment. The third and last point is concerned with simulation run results, which should be documented together with applied post-processing (i.e., analysis) steps. Altogether, MIASE focuses, particularly, on documentation and repeatability of experiments but also addresses a set of experiment tasks including the configuration of input parameters, model execution, data collection, and analysis.

Sargent [162] subsumes validation experiments, by distinguishing between comparing two models, and comparing model and underlying system. He proposes three relevant approaches for a comparison: graphical comparison (e.g., line charts, histograms, etc.), creation of confidence intervals, and hypothesis testing. All of these approaches deal with one specific experiment task: data analysis.

2.1.3. Life-Cycles and Workflows

Various life-cycles have been designed for the overall M&S process. Sargent [162] proposes such a life-cycle for generating credible models (see Figure 2.3). It deals with three principal subjects:

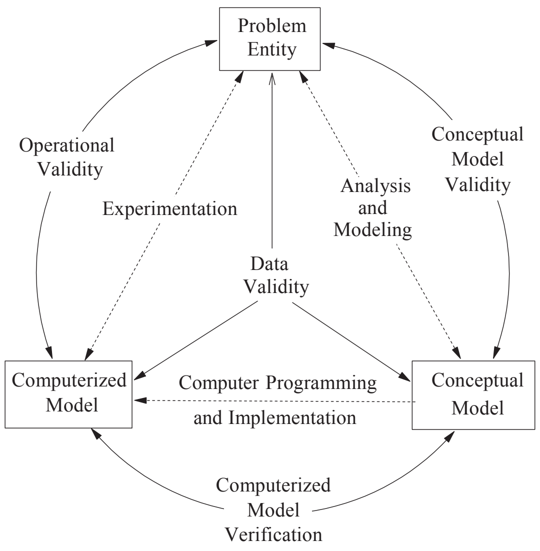


Figure 2.3.: The model life-cycle proposed by Sargent [162].

- *problem entity*: the system under investigation
- *conceptual model*: the mathematical/logical/verbal representation of the problem entity
- *computerized model*: the implemented model

By analyzing and modeling the problem entity, the conceptual model is generated and later implemented which leads to the computerized model. Further insights and knowledge about the problem entity are gained during experiments with the computerized model. All three subjects are closely related and have to be reconsidered multiple times during the process by applying *Verification and Validation (V&V)* methods. While this life-cycle does not give a deep insight into the experiment itself, it offers implications about experiment goals. Exploratory analysis or parameter fitting experiments can be conducted to gain knowledge about the problem entity, while operational validity can be achieved in validation experiments where data produced during simulation is compared to data produced by the problem entity.

Balci [12] describes an alternative life-cycle for model creation (see Figure 2.4), identifying the following steps that are also relevant for experimentation:

- *system and objectives definition*: the specification and definition of the questions to be answered System properties are identified that are of relevance for answering those questions.
- *conceptual model*: the description of the model with respect to the previously specified questions
- *communicative model*: a model that allows a collaborative investigation with other parties (e.g., researchers)
- *programmed model*: translating the communicative into an executable model
- *experimental model*: an experiment with all required parameters
- *simulation results*: the results of the experiment with respect to the raised questions

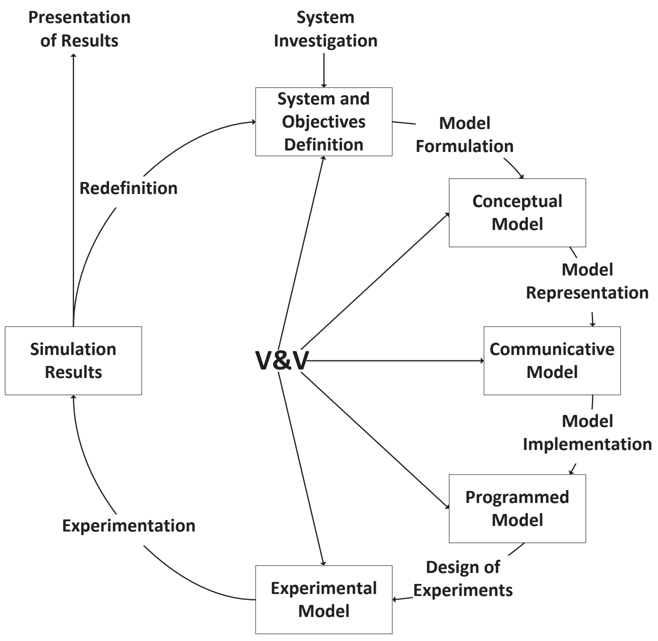


Figure 2.4.: A simplified variant of the model life-cycle proposed by Balci [12].

The steps are arranged circularly, i.e., after simulation results have been produced, a reconsideration of the system and objectives definition is required to evaluate whether it leads to desired results. All steps are interconnected through V&V procedures. The workflow offers similar implications about experiment goals, as the one of Sargent, and furthermore, underlines an important task — the specification of objectives. This is an essential task of the M&S workflow, and a similar step is required for experimentation, as the first task of each experiment is the translation of the experimental goal into a proper experiment specification. Balci references this issue by addressing an experimental model in the life-cycle, which comprises all required information to conduct an experiment.

In order to execute experiments, they have to be implemented in more concrete workflows. Such a workflow is described by Rybacki et. al. [160] for validation experiments (see Figure 2.5). It includes a theoretical analysis of the model for finding suitable validation methods that shall be used during the experimentation process. The experiment itself comprises four tasks: setup of the simulation (which refers to configuring model and simulator input parameters), model execution, data analysis, and evaluation of results.

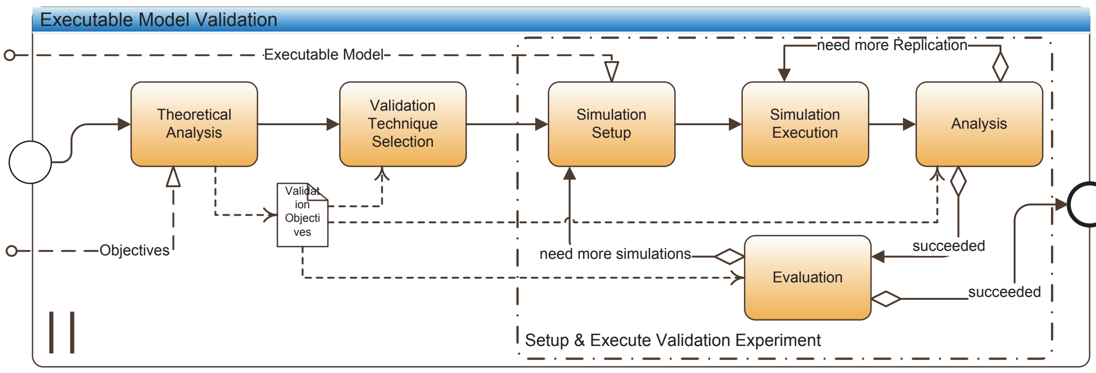


Figure 2.5.: Validation workflow described by Rybacki et. al. [160].

A realization of an experiment workflow can be found in the experimentation layer of JAMES II [83] (see Figure 2.6). Three components are responsible for controlling the experiment:

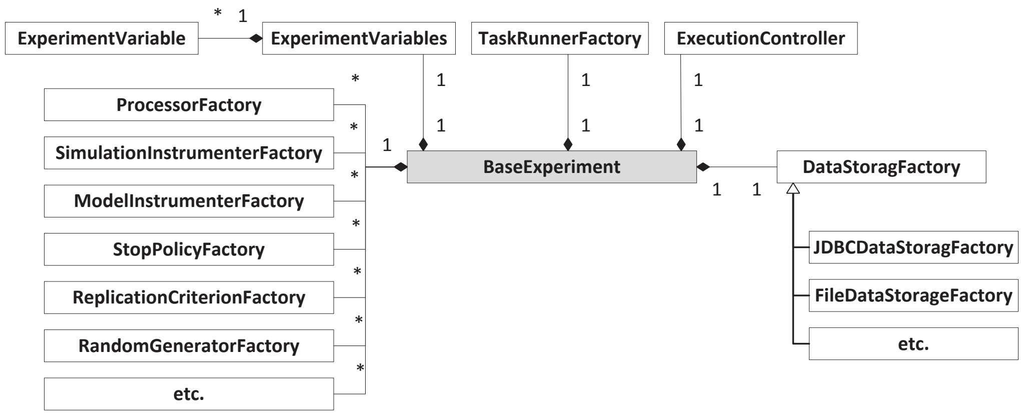


Figure 2.6.: JAMES II's experimental layer [83].

- *base experiment*: holds the model and prepares required data for experiment steering and to create random number generators, replication criteria, stop policies, instrumenters, and data storages
- *simulation runner*: responsible for executing simulation runs
- *execution controller*: realizes different simulation control schemes (e.g., automatic or interactive)

The execution controller handles organizational issues, while base experiment and simulation runner handle concrete experiment tasks, including input parameter configuration (through steering and random number generation), model execution, data collection (through instrumentation and handling of data storages), and data analysis (through replication criteria and run stop policies). Base experiment as well as simulation runner have to be configured properly by the user in order to fulfill their tasks.

Al-Zoubi and Wainer [2] propose a workflow system for managing and executing various workflow patterns on *Simulation Restful Interoperability Simulation Environments (RISEs) Servers* and present an example workflow for experiment execution. It starts with the configuration of model input parameters that are submitted to the simulation system, where simulation runs are executed until they reached a final state, met a stopping criterion, or produced an error. Afterward, simulation output data, or respectively error files, are processed. Depending on the results, the experiment instance is discarded, or maintained for further evaluation. Altogether, this workflow covers the tasks model configuration, model execution, data collection, analysis, and evaluation.

2.1.4. Experiment Goals

The overall experiment goal, as well as respective sub-goals have to be specified properly, as they influence the methods applied during the experiment, e.g., optimization as experimental goal suggests parameter search as model configuration method. Only the experimenter can specify such goals, since he decides what the desired achievements of the experiment are. A variety of such principal goals has been identified for M&S.

For instance, Barton [16] proposes a list of possible goals that are of relevance for experiments during the M&S life cycle. The list is sorted according to the point in the M&S life cycle, at which the goals are usually pursued.

- *early goals*: validation and screening of parameters.
- *middle goals*: sensitivity analysis, understanding the model, and predictive results.
- *late goals*: optimization and robust design.

Validation of models is an important step in the M&S life cycle which helps to answer the question whether the modeler was successful in "building the right model" [13, p. 41]. Therefore, simulation output of the model is compared to the behavior of the reference system which can be a real system or another model.

According to Kleijnen [115], screening can be seen as a part of sensitivity analysis which checks the sensitivity of models against changes in input parameters, by comparing simulation outputs of the same model, executed with different input parameter settings.

Understanding the model and producing predictive results can be achieved in exploratory experiments. They are usually conducted through parameter scans, however methods originally designed for optimization, e.g., parameter search algorithms, might lead to the goal, as well. In any case, the analysis of simulation results is used to gain insights into the modeled system.

Optimization experiments are used for improving the model (i.e., by optimizing the simulation output of the model itself) or the underlying system (i.e., by optimizing the simulation output of the model and applying the results on the system). In both cases, the model is executed with different input parameter settings and their simulation results are evaluated to find the optimal one.

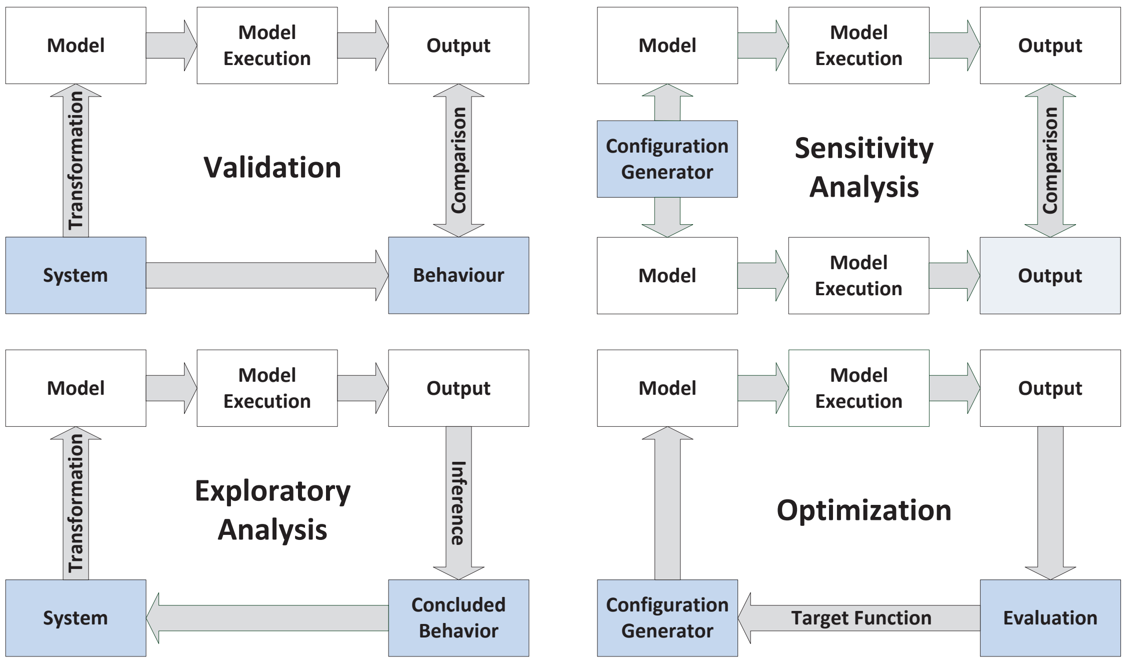


Figure 2.7.: Schemes for the different experimental goals. Note that these represent a very abstract view. In many experiments the goals cannot be strictly distinguished (only the validation scheme includes a comparison of model output and system behavior, however in reality, most experimenters refer their results to the real system in any case).

All in all, four principal experimental goals can be identified: validation, sensitivity analysis, exploratory analysis, and optimization (see Figure 2.7). Note that these are principal goals which sometimes are not distinguished clearly, e.g., only validation includes a comparison of model output and system behavior, however in reality, most experimenters relate their results to the real system even if they not specifically perform a validation experiment.

To pursue an experiment goal, different *sub-goals* have to be fulfilled, e.g., for optimization it is necessary to do a parameter search and to calculate objective functions. Those sub-goals influence the choice of used methods. For instance, an objective function is usually provided by analyzing simulation output data, e.g., through steady state estimation [7], simulation-based model-checking [53], or statistical methods [7]. The analysis relies on output data collected from simulation runs. Hence, simulation engines and data collection methods are required. For relating objective values to their respective input parameter settings (e.g., for optimization and sensitivity analysis) an evaluation step is required.

2.1.5. Gained Insights

Even though the discussed publications cover different points of view, they denote six general tasks that have to be considered during the execution of simulation experiments.

The most central task is *model execution* which is mostly referred as simulation, and directly or indirectly mentioned in each of the reviewed publications. Cellier's experiment definition includes the extraction of data from the model which includes executing it. Zeigler's experimental frame is centered around this task. Furthermore, Law and Kelton discuss simulation software and random number generation which is an essential component of many simulation algorithms. Kelton directly asks how to make the most efficient (simulation) run in his list of relevant questions, while the MIASE guidelines stress the importance of documenting simulation steps. All simulation experiment workflows comprise a step for executing simulation runs, and each of the discussed experiment goals rely on executing the model under investigation.

An additional task that all presented approaches deal with, is the *configuration* of model input parameters, i.e., the generation of input parameter settings. Cellier directly addresses input parameters, and Zeigler refers to them as initialization parameters in the list of information the experimental frame has to provide. Law and Kelton discuss the topic in their book, and Kelton highlights it in the list of questions to be considered during a simulation experiment (*"What model configurations should you run?"*). Furthermore, experiment workflows include steps for generating model parameter settings, for instance, during the simulation setup phase or through experiment steering. Finally, to pursue experiment goals, selecting input parameter configurations is a major concern, e.g., parameter search methods are essential for optimization and can be used for validation to find critical spots in the model's parameter space.

The next important tasks are *data collection* and *analysis*. Cellier talks about *"extracting data"* from a model, which can only be done by dealing with both tasks. The experimental frame includes observational variables and data collection methods which are used for collecting data during model execution, whereas stop criteria trigger analysis (the simulation is executed until sufficient data for analysis has been produced). Law and Kelton examine output data analysis for comparing parameter settings, and variance reduction techniques (both are tightly coupled to input parameter configuration). Furthermore, Kelton formulates three questions about analysis (*"How should you interpret and analyze the output?"*) and decisions to be made based on analysis results (*"How long should the runs be?" "How many runs should you make?"*). Worfklows use data collection and analysis methods for producing results, and both tasks are required to achieve experiment goals, e.g., to create objective values for an optimization experiment.

In many cases, an additional task is necessary for evaluating analysis results. During this *evaluation* task, data provided by analysis might be further processed for two reasons. First, analysis results might be used as feedback for creating additional interesting model input parameter settings (e.g., objective values in order to guide parameter search methods). Second, the final experiment results have to be generated, e.g., in form of a visualization.

In his modeling life-cycle, Balci mentions another very important task, the definition of relevant questions the model and, therefore, the experiment has to answer. This *specification* task is essential in order to translate the experimentation goals, the experimenter has in mind, into a description that can be used for executing the experiment. To tackle the different tasks a differentiation of the overall goal into sub-goals is required. For instance, if the goal is optimization, it is required to know what should be optimized, i.e., which objective function (influencing the analysis and evaluation task) or which bounds the input parameters should have (influencing the configuration task). The specification, hence, has to be oriented on experiment goals and sub-goals, and the resulting description should include information about which methods have to be applied for each task.

Figure 2.8 shows an overview over the different tasks and their dependencies. Input parameters

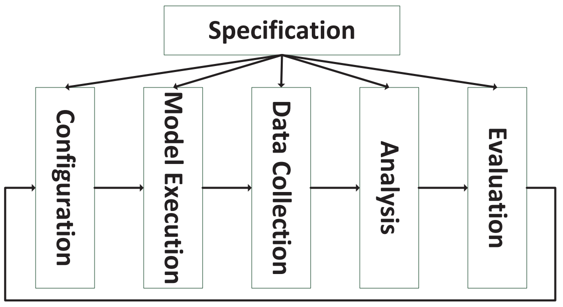


Figure 2.8.: Experiment tasks and their dependencies.

produced during configuration are generated and used for model execution. Data are collected during simulation runs, and analyzed, afterward. Analysis results are evaluated to produce feedback for configuration. All tasks depend on the specification of the experiment.

2.2. Experimentation in Simulation Tools

A variety of simulation tools, used in different application domains, exists. Some tools are centered around a specific task, like output analysis of simulation runs. For instance, AKAROA 2 [51] has been specifically designed to facilitate the execution of *Multiple Run In Parallel (MRIP)* [145], analyze the results, and control simulation replications and end times. Thereby, simulation runs are replicated on multiple machines, which are connected, e.g., in a network, and results are aggregated by statistical methods. Similar to STARS [135], and SIMPROCTC [43], both sharing a similar design, AKAROA 2 has been developed in (and is restricted to) the field of network research.

A more general framework is AUTOSIMOA [88], designed for the automated analysis of simulation output. It focuses on executing multiple simulation replications as well, but is not restricted to a specific M&S paradigm. The major goal is to assist users in finding good estimates for the amount of required replications and simulation lengths by considering, e.g., steady state estimation methods. Hence, it shares the overall aim of providing accurate and precise statistics of simulation output results with AKAROA 2, STARS, and SIMPROCTC — all tools deal with the execution and analysis of simulation runs.

However, none of them has been designed to cover the whole simulation experiment process, as all three tools lack, e.g., in providing methods for generation and evaluation of (model) input parameters. Alternative tools exist that offer features for complete simulation experiments. In the following, such tools will be discussed with respect to the tasks they support.

2.2.1. Tools for the Complete Experiment Process

To gain a differentiated overview over simulation tools, commercial (e.g., Arena, AnyLogic, SIMUL8, Simulink) as well as academic (e.g., SIMPLEX3, BIOCHAM, SBW, BETAWB, COPASI, SAFE) software used in different application domains (manufacturing, engineering, systems biology, network simulation, etc.) will be surveyed in this subsection. Thereby, the primary focus is on features for simulation experimentation. Hence, additional methods for modeling, analytical evaluation, or experiments centered around simulation algorithms, e.g., performance experiments, that are provided by some of the tools will not be discussed in depth.

Arena is a M&S tool created by the Rockwell Corporation [108], specialized in discrete event simulation, and used in professional domains like manufacturing, logistics, or business processes [31]. Besides powerful modeling tools, it also provides methods for executing simulation experiments.

It supports different experiment goals like optimization, sensitivity analysis, and exploratory analysis. The functionality of Arena is divided into different modules, which can be combined according to the experiment process. This includes methods for configuring input parameters, e.g., Arena Process Analyzer (providing various experiment designs), Arena Input Analyzer (providing statistical methods for analyzing model input parameters e.g., distribution fitting), or the external optimization library OPTQUEST [71] (providing methods for searching through parameter spaces). Furthermore, Arena allows to configure simulation runs, and defining replication criteria or simulation stop conditions. After the execution, simulation runs can be analyzed statistically. Evaluation is supported by modules for visualization or animation, and by parameter search methods that calculate objective values in order to search optimal input parameter setting.

All in all, Arena explicitly offers functionality for configuring model input parameters, model execution, as well as analyzing and evaluating results. The methods for each of these tasks and, therefore, the whole experiment can be specified and controlled by using the Arena Graphical User Interface (GUI).

AnyLogic is a M&S tool created by XJ technologies [31], used for discrete event as well as continuous and hybrid simulation. Its application areas include process analysis, logistics, manufacturing, health care, and military purposes, where various goals can be pursued, e.g., forecasting and strategic planning, process analysis and optimization, optimal operational management, or processes visualization. AnyLogic offers an extensive experimentation framework, providing methods for goals like optimization, or sensitivity analysis.

Similar to Arena, AnyLogic comprises components to realize experiment tasks that are specified by a GUI. It uses OptQuest [71] for parameter search, and provides means for input analysis with Stat::Fit [34]. Internal fitness functions create objective values used as feedback for parameter search algorithms. To execute simulation runs, stop conditions, replication criteria, and execution algorithms are available. Simulation results can be analyzed statistically and evaluated by generating charts and histograms.

Altogether, an experimentation process of AnyLogic comprises the tasks specification, model parameter configuration, model execution, analysis, and evaluation.

SIMUL8 is a discrete event M&S software, created by the SIMUL8 Corporation [33], that is mostly applied in production, manufacturing, and logistics, and usually used for exploratory analysis to test extreme conditions or investigate function and load of modeled production systems. It is, furthermore, possible to test the influence of parameters in sensitivity analysis, or to conduct optimization experiments.

The experiment process of SIMUL8 deals with the tasks specification (by using the SIMUL8 GUI), model parameter configuration, model execution, data collection, and analysis. For configuring a model, a fixed set of input parameter types is available, including cycle times, arrival and production rates, or capacities and statistics of production equipment. To generate interesting parameter settings, parameter scan and search methods are offered. SIMUL8 supports the control of simulation runs, e.g., by stop conditions, for which third party software can be integrated. Furthermore, model properties can be observed during the simulation run and results analyzed, afterward. Observable properties are, e.g., the use of production equipment, the analysis of bottlenecks, as well as different performance measures. A key feature of SIMUL8 is the ability to visualize results or animate the simulation process.

Simulink is a MATLAB package for modeling and simulation of continuous and hybrid systems [5]. It is mainly used in engineering and technical applications. Functionality can be extended by MATLAB programming or including additional packages.

Simulink ships with methods for pursuing different experiment goals, like validation, optimization, or exploratory analysis. The experimentation process is centered around the model, which can be parametrized by hand or by using according tools, e.g., the MATLAB Optimization Toolbox. Simulation runs can be parametrized as well, e.g., by choosing the desired numerical integration method or defining simulation stop conditions. Special components that collect data during execution can be integrated into the model, and results can be statistically analyzed through functionality provided by MATLAB.

All in all, the experiment tasks specification (through the MATLAB GUI), parameter configuration, model execution, data collection, analysis, and evaluation are explicitly supported by Simulink.

Simplex3 has been developed as an M&S tool for multiple application areas like science, economics, medicine, etc [164]. It supports continuous as well as discrete-event based models and is aimed at optimization and exploratory analysis. The specification of simulation experiments is done by using the *Experiment Description Language (EDL)* (see Algorithm 1 for a simple example) or a GUI.

A Simplex3 experiment comprises the tasks: experiment specification, configuration of model input parameters, model execution, recording and analyzing simulation output data, as well as evaluation of results. Model input parameters can be configured using parameter search and parameter scan functionality. For model execution, an exchange of the underlying simulation algorithm and associated components, e.g., integration method, step-size control, or random number generators, is possible.

Algorithm 1 Example description of a parameter search with SIMPLEX3-EDL, to find the optimal rate $r1$ for a model representing the chemical reaction $A \xrightarrow{r1} B$. The search is controlled by variable `rateConstant`, which holds the input value for $r1$ in each iteration of the search (line 14). Its initial value is 0.5. In each simulation run $1,000$ steps are executed. The amount of steps is given by variable `endPoint` (line 16). The output of the simulation is the count of species B, which is used as objective for the search (line 18). If the objective value is satisfying, i.e., `objective` is greater than 100, the experiment is finished (lines 20–23). Otherwise, it is continued with `rateConstant` (and therefore $r1$) being increased by 10 percent (line 25).

```
1  EXPERIMENT SearchExample
2
3  DECLARATION OF VARIABLES
4          rateConstant (INT) := 0.5
5          endPoint (INT) := 1000
6          threshold (INT) := 100
7
8  BODY OF EXPERIMENT
9
10 ...
11
12 LOOP <search> WHILE Continue REPEAT
13
14     <r1> := rateConstant;
15
16     SIMULATE TO, endPoint;
17
18     objective := <countB>
19
20     IF (objective >= threshold)
21     Do
22          EXIT <search>
23     END
24
25     rateConstant := rateConstant + (rateConstant * 0.1);
26
27 END LOOP <search>
28
29 END OF SearchExample
```

Model variables can be observed and analyzed by calculating running means, or frequency spectra. Finally, results are evaluated, e.g., with parametrized plots, line, or pie charts.

BioCham (Biochemical Abstract Machine) [52] is an M&S environment used in the area of systems biology. It offers a rule-based language for modeling biochemical systems. Experimentation goals like optimization, exploratory analysis, or validation of temporal properties of the model, can be pursued.

After the specification, by BIOCHAM's GUI or experiment language, an experiment consists of the tasks model parameter configuration, model execution, analysis of simulation output data, and evaluation. Different methods, like evolutionary or parameter scan algorithms, are provided for configuring the model, which can be executed with boolean, differential, or stochastic simulation engines. Simulation output data are collected as trajectories of species counts over time. By performing probabilistic model-checking, such trajectories can be checked for properties specified in temporal logic. Finally, BIOCHAM offers visual tools for evaluating the results.

SBW (Systems Biology Workbench) is a framework for modeling and simulation of biochemical reaction networks [96]. It comprises open interfaces for integrating external tools and additional components. Existing components for experimentation cover the goals exploratory analysis, sensitivity analysis, or optimization. Such experiments can be specified using a GUI or the *Simulation Experiment Description Markup Language (SED-ML)*.

The experiment process comprises the selection of model and simulator, model execution, observation of relevant model variables, and analysis. Different parameter settings are generated, e.g., through parameter searches. The simulation engine JARNAC, which can work continuously or in a stochastic, discrete-event way, executes models. Selected species are monitored along the way, and resulting trajectories can be analyzed using various analysis techniques, e.g., steady state estimation methods. As one requirement of SBW is code reuse, an integration of methods from other tools is possible for the different tasks.

Altogether, SBW supports the tasks model configuration, simulation, data collection, analysis, and evaluation.

BetaWB (Beta Workbench) is an additional tool from the realm of systems biology [40]. It allows the specification of stochastic models in the modeling language BLENX [39], and provides experimentation features to conduct, e.g., exploratory analysis.

The experiment process can be triggered by a command line, where model input parameters, generated by hand or external tools, can be set through input files. Different simulation engines are available, allowing, e.g., sequential and parallel executions that are controlled, e.g., by replication criteria and simulation stop conditions. Variables, species, and complexes are observed during simulation runs and results returned as files which can be analyzed and evaluated using the BETAWB Plotter or external tools.

Hence, an experiment with BETAWB includes the tasks model configuration, simulation, data collection, analysis, and evaluation.

COPASI (COmplex PAthway SImulator) is a platform-independent simulation tool for biochemical systems [92]. It supports the execution of tasks on different levels of the simulation experiment to realize, e.g., optimization as overall experimental goal or steady state mean estimation on the analysis level.

After an experiment has been specified with COPASI's GUI, the experiment steps model configuration, simulation, data collection, analysis, and evaluation are executed. Models can be configured by parameter scan or parameter estimation methods, and simulated by stochastic or deterministic simulation engines. Special focus is laid on analysis, as COPASI offers various methods for this task, e.g., steady state estimation, statistical analysis, or cycle detection. Visualization methods for evaluation of experimental results are provided, as well.

SAFE (Simulation Automation Framework for Experiments) is a tool specifically designed for simulation experiment support [148]. It includes a sophisticated experiment structure and includes a central database facilitating the data-flow. Experiments can be specified in an eXtensible Markup Language (XML) based format.

After interesting input parameters for the model have been identified, through a 'design point generator', simulation runs are executed using the NS-3 simulator [142] — the third iteration of a series of discrete-event network simulators [142] which has been designed as an open environment for research and teaching in the field. Different execution schemes, e.g., MRIP are supported for that. Output data are specified by the user beforehand, logged during execution, and returned as event trace. Afterward, they can be stored in a database, statistically analyzed, or plotted.

All in all, the overall experiment process of SAFE tools comprises the tasks, experiment specification, model configuration, model execution, data collection, analysis, and evaluation. Different user profiles (novice and power users) are distinguished to adjust support mechanisms that assist in the configuration of parameters, the simulation on different architectures (e.g., distributed), and data analysis.

2.2.2. Common Tasks

Table 2.1 gives an overview over the reviewed simulation tools, and lists selected techniques used to realize the experiment tasks identified in Section 2.1. Despite the variety of tools, most of these experiment tasks are addressed in each tool explicitly.

The central task, handled by all of the tools, is of course model execution. Several tools offer different simulation strategies and algorithms, e.g., Simulink which allows different integration methods, or BioCham and SBW supporting continuous as well as stochastic simulation. Besides providing the right algorithm for the given modeling formalism, the reason for different simulation engines can also lie in efficiency which, for instance, can be acquired through parallelization (e.g., in BetaWB). No matter the reason for realizing different simulation engines, choosing the appropriate one is an important topic. Furthermore, additional methods that control simulation runs, like stop conditions or replication criteria (that also trigger analysis), are widely supported.

Each of the presented simulation tools offers the opportunity to configure model input parameters. In addition, most tools provide methods for doing this automatically by parameter search or DOE algorithms (e.g., Simulink, BioCham, SAFE), or have extensions points facilitating the integration of such methods (e.g., OptQuest in Arena and AnyLogic).

In order to create proper results, all tools include at least one method for the analysis of simulation output. This is necessary to gain valuable information about the executed model, as raw simulation data are usually hard to interpret. All tools provide classical statistical methods (e.g., steady state mean estimation, calculating means, variance, or confidence intervals) for analysis. However, also more specialized techniques are available e.g., statistical model-checking in BioCham. Several tools, e.g., Arena and Simulink, use analysis methods to set simulation stop conditions or replication criteria.

Observation of simulation runs and the collection of data is not explicitly considered in every tool. However, as this step is necessary to generate input data for analysis, each tool deals with this task, at least implicitly. Furthermore, some tools support explicit concepts for data collection, e.g., SAFE, or the observation of specific model components during runtime, like species in BioCham. This is beneficial to collect the right information in order to perform analysis efficiently on relevant data only, and to save memory.

For evaluating experiment results, all tools offer graphical methods, e.g., charts and plots. Furthermore, evaluation methods are often tightly connected to the applied configuration method. For instance, many configuration methods need evaluation results as feedback, e.g., the objective value of a parameter search algorithm is a typical evaluation result.

All of the tools offer user interfaces for specifying experiments, like a command-line e.g., (BetaWB or SAFE) or GUI (the other tools). Furthermore, some tools provide advanced features for experiment specification, like the domain-specific language Simplex3-EDL. Such instruments facilitate the translation of the experiment goal into an appropriate specification in order to communicate to the experimentation tool what has to be done, i.e., which techniques have to be executed at which point.

2.3. A Common Simulation Experiment Structure

Even though terminology and viewpoints differ, the reviewed literature and tools for simulation deal with similar tasks, independently from goal or application area. The process of applying methods for such tasks can be described in a common experiment structure (see Figure 2.9) that helps users in getting aware of the tasks they have to consider for executing an experiment.

On top of the structure resides the overall experiment goal, e.g., optimization, validation, or exploratory analysis. The experiment structure itself comprises six tasks: specification, model configuration, model execution, data collection, analysis, and result evaluation, which have to be considered in experiments, independently from the concrete goal. In order to fulfill an overall goal, adequate

Tool	Specification	Configuration	Model execution	Data Collection	Analysis	Evaluation
Arena	•GUI	•parameter search •experiment design •distribution fitting	•discrete-event	•implicitly (depending on analysis)	•statistical	•visualization •feedback
AnyLogic	•GUI	•parameter search •experiment design •distribution fitting	•discrete-event •continuous •hybrid	•implicitly (depending on analysis)	•statistical	•visualization •feedback
SIMUL8	•GUI	•parameter search •parameter scan	•discrete-event	•observation of specific variables	•statistical	•visualization •feedback
MATLAB/ Simulink	•GUI	•parameter search •parameter scan	•continuous •hybrid	•observation of specific variables	•statistical	•visualization •feedback
Simplex3	•GUI •EDL	•parameter search •parameter scan	•different algorithms and components	•observation of of specific variables	•statistical	•visualization •feedback
BioCham	•GUI experiment language	•parameter search •parameter scan	•boolean •stochastic •continuous	•trajectories of specific variables	•statistical •model-checking	•visualization •feedback
SBW	•GUI •SED-ML	•parameter search •bifurcation	•continuous •stochastic •discrete-event	•observation of specific variables	•statistical •steady state estimation external tools	•visualization •feedback
BetaWB	•GUI •command-line	•external tools	•stochastic •sequential •parallel	•observation of specific variables	•statistical •external tools	•external tools
COPASI	•GUI	•parameter scan •parameter search	•stochastic •deterministic	•implicitly (depending on analysis)	•statistical •steady state estimation •cycle detection	•visualization •feedback
SAFE	•GUI •XML-based language	•Design Of Experiments (DOE)	•discrete-event •different execution schemes (e.g., MRIP)	•observation of specific variables	•statistical	•visualization

Table 2.1.: Simulation tools and selected techniques to support experiment steps. Red boxes indicate that the experiment step is not or not explicitly supported by the tool. Yellow boxes indicate explicit support but a restricted set of available techniques. Green boxes indicate explicit support and flexible integration of arbitrary techniques of the experiment step.

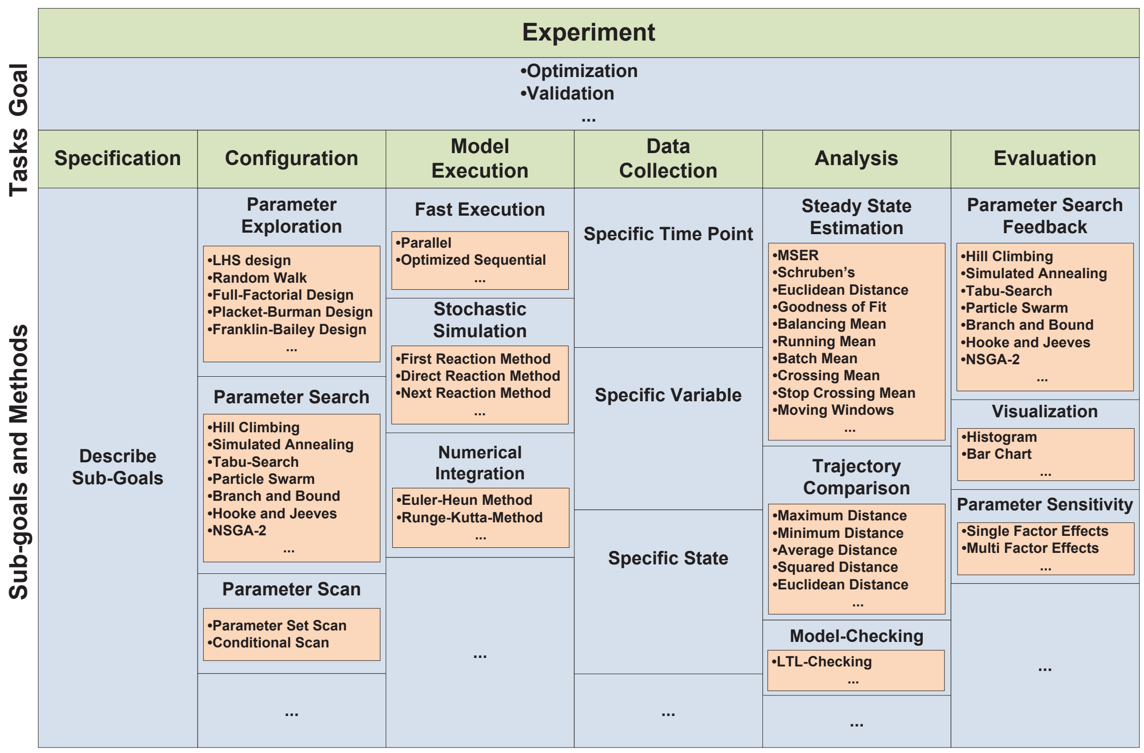


Figure 2.9.: The organizational structure of a simulation experiment. Components in green are obligatory, components in blue and orange have to be selected by the experimenter. Goals and sub-goals (blue) depend on the desired outcome of the experiments. Methods (orange) depend on the according sub-goal and additional factors — they might be selected automatically.

sub-goals have to be achieved for each task (for specification, just one relevant sub-goal exists, the description of the experiment). Achieving the overall goal results from a composition of such sub-goals, e.g., to realize an optimization, the sub-goal of the configuration task is usually a parameter search. In most cases, a variety of methods is available for each sub-goal (except for observation, where the method is, in most cases, directly defined by the sub-goal, e.g., tracking a specific variable).

It is mandatory to distinguish between the overall goal of the experiment and the tasks that have to be conducted for achieving it. A similar distinction is required regarding the sub-goal to be pursued in each task, and the concrete methods applicable for reaching the sub-goal. For instance, the overall goal of an experiment can be optimization, whereas corresponding sub-goals might include parameter search for the configuration task and reaching a given confidence interval in the simulation results for the analysis task. Calculating confidence intervals can be achieved by applying a statistical method like a student-t test. Analysis results are then used to achieve reliable responses for a parameter search method like an evolutionary algorithm. Both sub-goals, parameter search and reaching a given confidence interval, are required to pursue the goal of optimization, but maybe also used for different goals, like validation.

Consequently, a tool that shall support users in general simulation experiments, should be oriented on the different experiment tasks, but arrange methods flexibly to achieve different goals and sub-goals.

2.3.1. The Six Tasks of a Simulation Experiment

Six typical tasks could be identified in the surveyed literature and simulation tools,. These tasks have been, previously, discovered for validation experiments [121, 125], but are relevant for experiments in general as discussed in sections 2.1 and 2.2. In the following, their role in the overall experiment process shall be illustrated.

Specification Specification is an essential task in each simulation experiment, as it connects the overall experiment goal with the concrete experiment execution. Ideally, it results in a formal representation of the goal, which is the foundation for all the following tasks. consequently, methods used to execute these tasks have to work according to the specification.

Since, the goal of an experiment is usually defined by the user (experimenter), the specification works as an interface between user and experiment. Tools, like a GUI or a domain specific language for experimentation, can facilitate the task and have to be translated into a specification that is understandable for the components that realize the experiment. The specification format has to follow the general tasks of a simulation experiment but provide flexibility with respect to goals and methods. Flexibility with respect to goals means that specifications can represent different goals, like optimization, validation, exploratory analysis, etc. Flexibility with respect to methods means that different methods can be specified, e.g., for analysis it should be possible to specify methods for steady state estimation as well as trajectory comparison, cycle detection, etc.

Finally, the specification is central for an essential requirement of scientific work: repeatability. Each specification should include all required information for repeating the specified experiment, if necessary.

Configuration of Model Parameters The second task is the configuration of model input parameters, which corresponds to generating test cases in software testing. During this task, those points in the model's parameter space are selected that shall be investigated during the course of the experiment. The parameters depend on the model, e.g., they could be primitive data types like double values for rates of chemical equations, or complex components like behavioral patterns in agent-based models.

Various methods for a systematic creation of parameter settings exist, ranging from parameter scans, over experiment designs, to parameter search methods. The latter relies on a loop of generating parameter settings and processing feedback about them. Many of the parameter search methods have been designed for the use in optimization experiments, e.g., evolutionary algorithms [6], which is why they are often called *optimization algorithms*. However, as they provide very useful functionality, they can be very helpful to achieve other goals as well, e.g., validation, by searching parameter settings that violate a given threshold. Thus, while parameter configuration methods, like parameter search algorithms, are often considered to be created for one specific goal, like optimization, a clear distinction between both (method and goal) is mandatory, and facilitates a flexible application of methods to different overall goals.

Model Execution Executing the model, i.e., performing a simulation run, is done by simulation algorithms that have to be accurate and efficient. Accuracy is required to produce trustworthy simulation results for analysis, while efficiency is needed, to produce results in reasonable time. Usually, a trade-off between accuracy and efficiency exists. A typical example for this is numerical integration, where methods, i.e., simulators, of different orders exist [30]. A higher order numerical integration method requires many time-consuming function iterations, but produces usually more accurate results than a lower order method that requires fewer function evaluations and is, therefore, faster. Hence, for each simulation run, a simulation engine has to be selected that provides the most suitable trade-off between efficiency and accuracy.

To obtain adequate flexibility in selecting the simulation engine, the exchange and parametrization of engines and their components has to be supported. This means that the general execution scheme (e.g., parallel or sequential) as well as relevant components (e.g., random number generators or event queues) have to be exchangeable. Besides efficiency and accuracy, this also helps in finding bugs, as comparing the results of similar simulation runs with different simulation engines can help identifying side effects.

The flexible integration of simulators leads to effort that has to be put into selecting them. In addition, some tools include further simulation parameters to be set, e.g., stop conditions or replication criteria which depend on the analysis of simulation results (the simulation has to be continued until the analysis is finished).

Data Collection Data has to be collected, during simulation runs, for further processing. The challenge of this task is to collect as much information as necessary to allow a proper analysis of the results, but as little information as possible to save memory and computation time during the analysis. This is especially important for complex models comprising many state variables, where monitoring all variables would lead to much produced data and high overhead resulting from the monitoring process, e.g., by listener calls for each changed variable.

The selection of required data highly depends on the following analysis, e.g., for realizing a steady state estimation, the discrete time points of the relevant variable have to be collected, whereas collecting data about other variables would be a waste of resources. Furthermore, a translation of the raw data, collected during simulation, into a format understandable by the analysis method is necessary.

Analysis Analysis is a complex part of simulation experiments, which is usually applied in two steps.

During the first step, called *single-run analysis* in the following, the results of single simulation runs are analyzed, e.g., by estimating a steady state statistic, or by checking properties formulated in linear temporal logics. If interest lies in the final state of the simulation run only, this analysis step is not necessary.

In the second step, called *multi-run analysis* in the following, the results of multiple replications are aggregated. The aggregation can be very simple, e.g., calculating the mean of the single-run results, or complex, e.g., when calculating monte-carlo variability. The analysis of replications is only required for stochastic simulations.

Analysis also influences the simulation execution, as the length of the simulation runs depends on the first analysis step — each run has to produce as much data as the analysis requires. Similarly, the number of required replications depends on the second analysis step — simulations are replicated as long as the analysis requires data.

Evaluation Evaluation, the final task of a simulation experiment, uses analysis results for two purposes.

On the one hand, feedback is produced for the configuration task, in order to identify additional interesting parameter combinations. This is, for instance, necessary for parameter search methods, where the analysis results represent the objective values of the executed parameter settings.

On the other hand, experiment results are generated. Examples for this are an optimal parameter setting, parameter sensitivity, or the visualization of analysis results.

2.4. Summary

This chapter reviewed literature (Section 2.1) and tools (Section 2.2) dedicated to simulation experimentation. Based on this review, an organizational structure has been derived (Section 2.3), and six typical tasks — specification, configuration, model execution, data collection, analysis, and evaluation — identified. The diversity of methods and strategies applicable for these tasks [163, 10, 115] demands an *explicit* and *flexible* addressing of them.

An experiment task is handled explicitly, if the user is aware that this task has to be considered during an experiment and at which stage of the experiment process it is executed. The explicit handling is necessary to help users in finding and selecting the most suitable methods in each stage to pursue the desired experiment goal.

Experiment tasks are handled flexibly, if the exchange and integration of different methods to fulfill tasks is supported. Flexibility is required to facilitate the realization, integration, evaluation, and use of diverse methods to provide sound implementations of current state of the art experiment techniques.

Flexibility can be achieved through a plugin based simulation tool, that allows the integration of arbitrary methods into the experiment process [82]. Such a tool is JAMES II [81], which is also general with respect to used modeling formalisms and application domains, making it an ideal base for a tool that guides users through different types of experiments. While not all experiment tasks are *explicitly* supported by JAMES II, so far, they can be easily integrated by developing new plugin types. The following chapter will show how the experimentation layer of JAMES II is extended.

3. Integrating the Experiment Structure into the JAMES II Experimentation Layer

"All life is an experiment. The more experiments you make the better."

Ralph Waldo Emerson

The previous chapter identified six tasks to be considered in a of simulation experiment and argued that those tasks have to be realized explicitly and flexibly to provide proper guidance. JAMES II [81, 85], a Java [75] based M&S framework, can be used to achieve this goal. Its plug 'n simulate concept allows reusing parts of algorithms, extending, configuring, and evaluating them. So far, JAMES II focuses mainly on integrating various modeling formalisms as well as associated simulation engines and components [49, 84, 124, 122, 103]. With the integration of JAMES II's experiment layer [83] a first step towards a unifying plugin based experiment tool has been developed. However, the experiment layer focuses primarily on the technical execution of experiments (e.g., parallel and distributed execution by setting a task runner, experiment control, data storage, etc.), and tasks model configuration, model execution, and data collection (see Figure 3.1). Other tasks like analysis are only triggered implicitly through stop and replication criteria. Especially the data flow between components (e.g., forwarding analysis results to the evaluation) is completely left to the user. GUISE shall offer this support in a plugin based experimentation environment, by extending JAMES II' experiment layer.

A first step toward this environment is realized in this chapter. The plugin system of JAMES II is exploited to realize experiment tasks and its experimentation layer is extended to incorporate them. Example experiments show flexibility and applicability of the approach for realizing various experiment types.

3.1. Using Plugins to Realize Experiment Tasks

To explain the plugin based experimentation in GUISE, the first part of this section gives an overview over the plugin system of JAMES II which has been developed by Himmelspach and Uhrmacher [85]. The second part explains how plugins are exploited to realize the six experiment tasks identified in Section 2.3.1, pp. 25.

3.1.1. Overview over the Plugin System of JAMES II

Most of the functionality of JAMES II has been integrated as plugins, so far, including modeling formalisms, simulation engines, event queues, random number generators, etc. Each plugin ships with a factory, that creates the objects, providing the concrete algorithm. This complies with the *factory method*, which is a typical software design pattern [63].

Plugin Type All plugins with the same purpose, are associated to a *plugin type*, e.g., all plugins realizing a simulation algorithm belong to the *processor* plugin type. A plugin type comprises the three components:

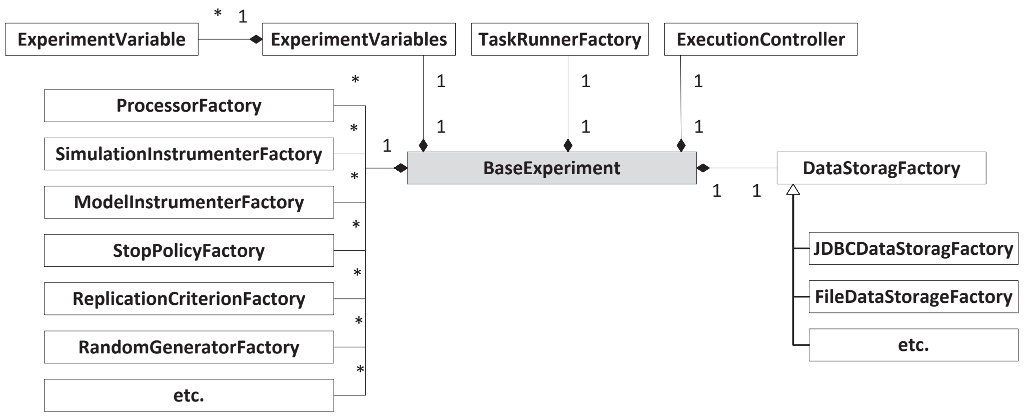


Figure 3.1.: JAMES II's experimentation layer [83].

- *abstract factory*: responsible for filtering plugins that comply to the actual plugin type
- *base factory*: has to be extended by the plugin factories belonging to the plugin type
- plugin description: an XML file that comprises the name of the plugin type, abstract and base factory, as well as meta-information

This list of plugin types can be easily extended by providing a corresponding XML file, abstract, and base factory.

Plugin Description A plugin itself is described by an XML file as well. It comprises the plugin's name and a list of factories which can be used for creating the plugin's functionality, e.g.,, if the plugin belongs to the processor plugin type, each factory in this list is able to instantiate a simulation algorithm. As mentioned before, the factories have to extend the base factory of the associated plugin type.

Plugin Detection and Application Plugins can be provided as standalone class files or jar files. They are registered at the central *registry* component, which follows the *singleton pattern* [63]. At each start-up of JAMES II, the registry loads, categorizes, and organizes the plugins in separated lists, according to their plugin types. Those lists hold the plugin's factories that extend the base factory, defined in the XML file of the according plugin type. Only those factories valid for the particular plugin type are considered during the plugin selection process, e.g., for the processor plugin type, only those factories that instantiate simulation engines and extend the according base factory are put into the same list. If an invalid plugin for the given plugin type has been selected, the registry automatically rejects the plugin.

If the user wants to define the kind of required functionality (e.g., any simulation engine for the given model) but not the concrete plugin, he has to provide the class of the *abstract factory* offering the desired functionality, and a set of parameters as parameter block (see next paragraph). The filtering process depicted in Figure 3.2 is, then, triggered. The given abstract factory applies filter criteria on the list of loaded plugin factories. Factories that do not match all criteria, are removed from the list. Filter criteria can be integrated as plugins as well, which ensures flexibility of the selection process. A typical criterion for processor plugins is the formalism of the executed model, as simulation algorithms are usually built for a specific model formalism. If no factory remains after the

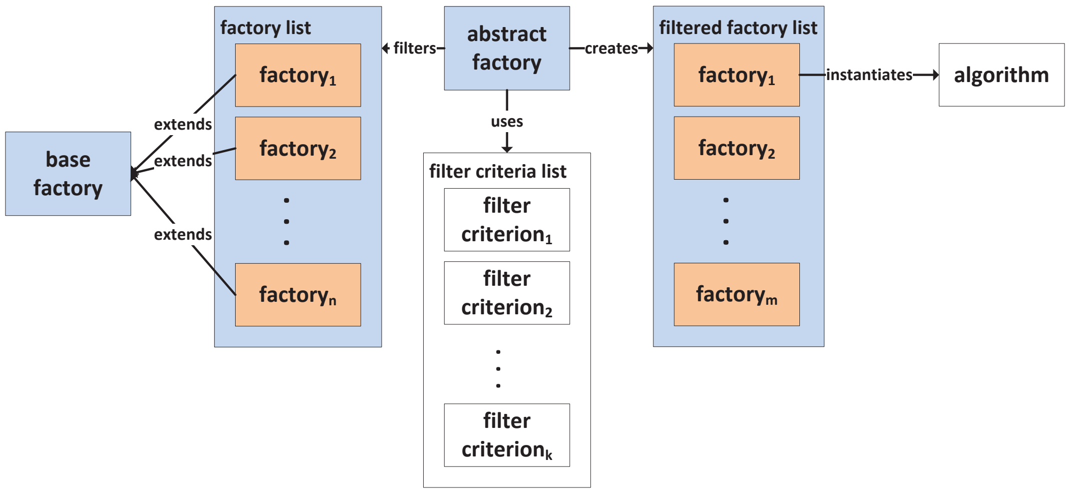


Figure 3.2.: The plugin detection and application process.

filtering process, an exception is thrown, stating that no factory can be found for the given setting. Otherwise, the first *factory* in the list will be returned, and used to instantiate the class providing the concrete functionality. Usually, those classes implement an according interface, like the `IProcessor` interface for the processor plugin.

Defining Plugin Settings with Parameter Blocks The plugin settings and parameters are defined in *parameter blocks*. Each parameter block is labeled with an identifier, holds a value associated to the identifier (values can be arbitrary objects), and contains a set of sub-blocks, being parameter blocks themselves. This design facilitates the creation of parameter settings for hierarchical structures, by associating each level of the structure with the corresponding level of the parameter block. A typical example for this are nested plugins, i.e., the plugin on the higher level has components that are realized as plugins as well. For instance, a parameter block for configuring a simulation algorithm using an event-queue and a Random Number Generator (RNG), comprises a label denoting the simulation algorithm's processor factory, as well as sub-blocks for configuring event-queue and RNG, which can hold sub-blocks for sub-components themselves.

By using parameter blocks, parameter settings can be described in arbitrary detail. This can be exemplified at the configuration of a method comparing a simulation output time series and a reference time series, e.g., by calculating the maximum distance (see Figure 3.3). The topmost block comprises the factory of the maximum distance method. Furthermore, a sub-block exists containing the reference time series. In many cases, an interpolation of the simulation output time series is required. The corresponding interpolation algorithm can be specified in an additional sub-block referring to its factory. Interpolation algorithms might need further sub-methods or parameters that can be defined in own sub-blocks, as well. This small example alone leads to three levels of detail: the comparison method, the interpolation method, and the parameters of the interpolation method.

A detailed description is beneficial for repeatability, as users can exchange parameter blocks and reuse them to repeat experiment and compare results. Furthermore, documentation is facilitated, as the configuration of the methods is reflected in parameter blocks.

3.1.2. Experiment Task Plugins

The plugin system of JAMES II is exploited, to integrate methods realizing the six experiment tasks, in a flexible and extensible manner. To manage the behavior of methods, they have to implement ac-

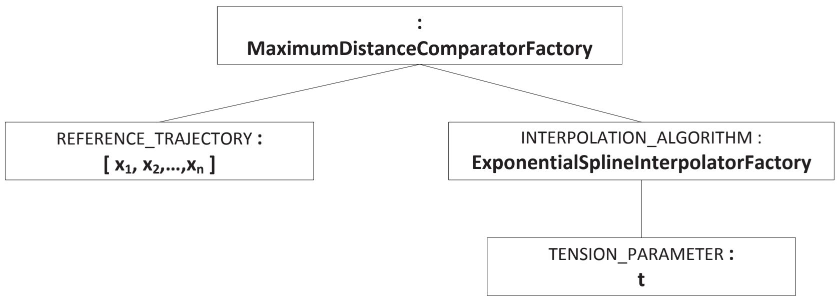

:
MaximumDistanceComparatorFactory
REFERENCE_TRAJECTORY :
[x_1, x_2,…,x_n]
INTERPOLATION_ALGORITHM :
ExponentialSplineInterpolatorFactory
TENSION_PARAMETER :
t

Figure 3.3.: Example parameter block for a trajectory distance comparator. The root contains the name of the used factory as value (without label). One sub-block contains the reference trajectory (denoted by the corresponding label) as list of numbers. The second sub-block contains the factory of the used interpolation algorithm which uses a tension parameter that is stored in a sub-block of the sub-block.

cording interfaces reflecting the methods role in the experiment process, e.g., as configuration method, simulation algorithm, etc.

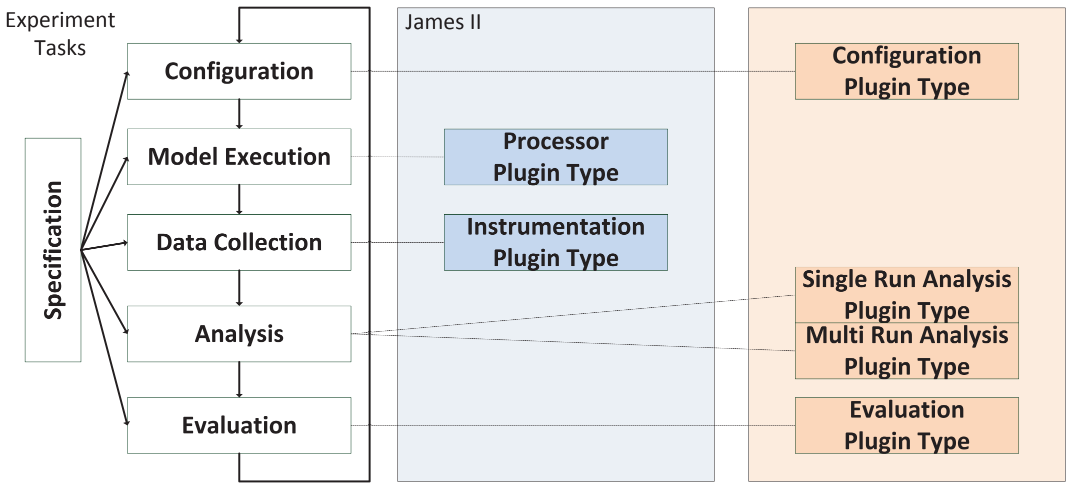


Figure 3.4.: Plugins for experiment tasks, that exist in JAMES II (blue) and that have to be realized (orange).

Five of the six tasks (configuration, model execution, data collection, analysis, and evaluation) have been integrated by at least one plugin type and interface. Figure 3.4 shows the implemented plugin types. Two tasks are already covered by plugin types existing in JAMES II —model execution and data collection.

The processor plugin type is a straight forward realization of the model execution task, and provides simulation engines. The according interface is the JAMES II `IProcessor` interface, which offers the `nextStep` method, for executing a model execution step.

The instrumentation plugin type instruments models or simulation engines with observer objects corresponding to the the observer pattern [63]. An observer implements the `IObserver` interface, providing the method `update` which is triggered when changes in the observed `entity`, in this case the simulation engine, happen. To make data collection more easily accessible to users, JAMES II offers an instrumentation language [80], for specifying the data that shall be collected.

As described in Section 2.3.1, pp. 25, analysis usually comprises two steps: the analysis of single simulation runs and the analysis of a set of replications. Both steps have to be reflected in own plugin types, as they are treated separately during the experiment. Hence, two plugin types are realized and each analysis type requires an own interface. Both interfaces comprise the method `analyse` which executes the analysis process on the given simulation output data and returns the corresponding result. They only differ in their return type. In the case of the `IMultiRunAnalyzer` interface, the return type is an integer value denoting the amount of required replications, whereas for the `ISingleRunAnalyzer` interface, the return type is a boolean value denoting whether an additional model execution step has to be processed. For integrating the analysis plugin types, special stop and replication criteria have been implemented. Both are controlled by methods implementing the respective analysis interface.

JAMES II already integrates functionality for handling the configuration task through experiment variables that can define input parameters of models. However, no plugin type exists for this functionality, so far. Furthermore, no explicit infrastructure for evaluating data, e.g., for processing feedback to the configuration, is provided. Thus, configuration and evaluation are each integrated by an own plugin type and interface. The `IConfigurator` interface provides a method `generate`, which iteratively generates parameter settings by taking received feedback into account (e.g., executed parameter settings and their objective values). The `IEvaluator` interface comprises the method `evaluate`, which works on analysis results of executed simulation runs and produces evaluation results.

The flexibility that lies in the plugin use can be seen at the various methods that have been integrated for GUISE, so far (see Appendix A, pp. 127). Note that all of them can be combined with arbitrary modeling formalisms and simulation engines that are implemented for JAMES II (which currently ships with more than 10 formalisms).

3.1.3. Realizing the Specification Task

The specification is the only task that is not handled by one or more plugin types. The reason for this lies in the fact, that specifying an experiment is concerned with translating the users desires and intentions into an experiment description format that can be used to configure and execute the subsequent experiment tasks. As the user's desires and intentions are initially 'stored' in the human mind, an automatic or algorithmic translation into the resulting format is hardly possible. The field of *Human Computer Interaction (HCI)* [101] deals with investigating the conditions and design of suitable tools for this task, like GUIs, or domain-specific languages (e.g., Simulation Experiment Specification via a Scala Layer (SESSL) [50]). While this work does not deal with designing such tools, parameter blocks are proposed as intermediate format between experiment specification and experiment execution.

Parameter blocks are an obvious choice for representing specifications, as they are used to configure plugins (see Section 3.1.1) and, therefore, all experiment tasks that are represented by plugin types. As described before, they are suited to store the most detailed description of an experiment setting, due to their adaptable, flexible, and hierarchical structure. Furthermore, they can be used as input parameters to control the experiment process. A parameter block specification contributes to repeatability and documentation, as users can exchange and reuse them to repeat experiments and compare their results.

While parameter blocks are very flexible and beneficial for specification, their creation by human users is not very convenient. Hence, a more accessible tool is still required, e.g., a GUI where parameter blocks are visually designed, or an experiment language whose expressions can be translated to parameter blocks.

Nevertheless, they make a systematic execution of different experiment specifications possible, by searching over the parameter block space (see Figure 3.5), for which existing parameter search methods might be used. The integration of those interfaces and methods is easily possible, by putting them at the beginning of an experiment and using resulting parameter blocks as experiment input.

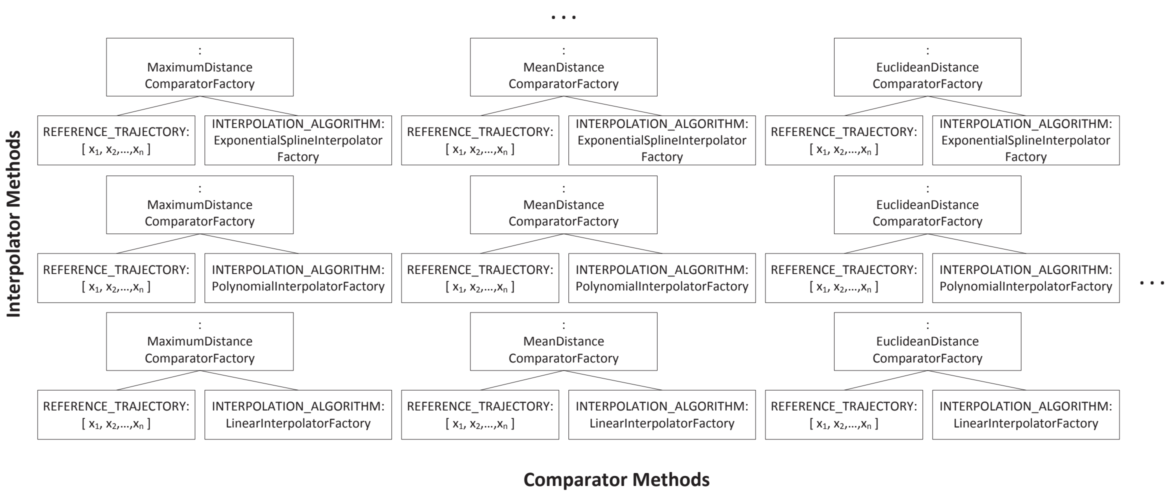

...
: MaximumDistance ComparatorFactory
: MeanDistance ComparatorFactory
: EuclideanDistance ComparatorFactory
REFERENCE_TRAJECTORY: [$x_1, x_2, \ldots, x_n$]
INTERPOLATION_ALGORITHM: ExponentialSplineInterpolator Factory
INTERPOLATION_ALGORITHM: PolynomialInterpolatorFactory
INTERPOLATION_ALGORITHM: LinearInterpolatorFactory
Interpolator Methods
Comparator Methods

Figure 3.5.: Example for a two-dimensional parameter block space, containing specifications for trajectory comparison methods, i.e., the analysis task. By moving horizontally through the parameter block space, the comparison method is changed, whereas the interpolation algorithm is changed by moving vertically.

3.2. Layered View on Simulation Experiments

Besides deciding which components have to be integrated as plugins, it is also necessary to define the interaction of such plugins. Usually, the six tasks of a simulation experiment are not arranged in a sequential manner, as loops exist in the execution scheme of an experiment. Such loops result from the repeated execution of methods which depend on the results of subsequent methods, e.g., the repeated creation of model parameter settings needs response from model execution, analysis and result evaluation. Those loops are nested, e.g., the loop of handling model parameter settings comprises a loop for executing those settings. Different layers in the experiment execution scheme can be identified, based on nested loops. See Figure 3.6 for an overview over these layers, which are explained in the following.

Experiment Layer The topmost layer includes the specification of the experiment, which influences all subsequent tasks, as it contains the information for configuring the according methods. Thus, all of the following layers rely on this task. By executing different experiment specifications, it is possible to explore the specification parameter space. This is beneficial if different experiment methods shall be tested, e.g., different analysis methods, while the rest of the specification remains unaltered.

Multi-Configuration Layer The second layer focuses on creating and executing a model with different input parameter settings. Consequently, the configuration plugin type is situated in this layer. Parameter settings generated by this plugin are executed in the subsequent layer, and the results are evaluated, afterward, by the evaluation plugin. As described before, feedback (e.g., objective values for executed parameter settings) may be created by the evaluation plugin, and forwarded to the configuration plugin, if necessary. This leads to a loop of parameter configuration, execution, and evaluation. The loop is finished as soon as the evaluation method has received enough data, or the configuration method is not able to produce any more parameter settings (e.g., due to given boundaries, or fulfilled cancel criteria).

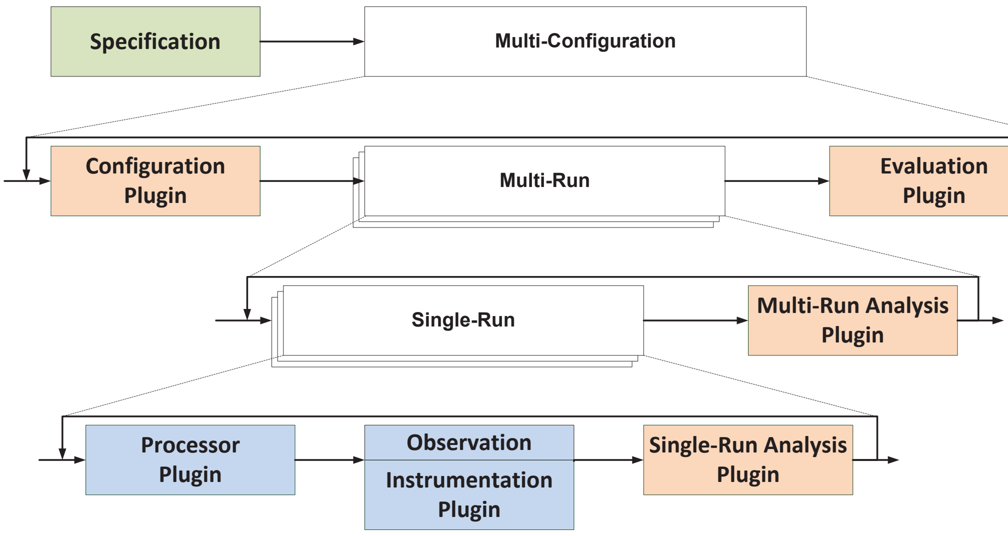


Figure 3.6.: Layered view on the six tasks of a simulation experiment with plugins.

Multi-Run Layer The third layer focuses on the replications of simulation runs with a given model parameter setting (that has been generated in the overlying layer). Replications are executed and analyzed with the multi-run analysis plugin. After each analysis step, the according method determines how many additional replications are required. This information is used as feedback to create the according number of replications. The loop of executing and analyzing replications is finished, as soon as the analysis method has sufficient data, or the given cancel criteria are met. Results of the analysis method are used in the multi-configuration layer as input for the evaluation method. The creation and execution of simulation runs is done in the subsequent layer.

Single-Run Layer The bottom layer comprises the execution of a single simulation run, the according data collection, and the analysis of collected data. JAMES II's processor plugin is used for execution, while simulator and/or model are instrumented with observer objects using the instrumentation plugin. Those observers are responsible for collecting data during the simulation run by monitoring changes in the observed entity. Collected data are forwarded to the single-run analysis plugin. After each analysis step, the method determines whether additional data, and therefore simulation steps, are required. In the case of more required data, the next simulation step is triggered. Otherwise, or if the model reached an end state, the simulation run is finished and the results of the single-run analysis are forwarded to the multi-run analysis in the multi-run layer.

Relation to the Experimental Layer of JAMES II Figure 3.7 illustrates the relation between the implemented plugins, realizing experiment tasks, and the experiment layer of JAMES II. Configuration and evaluation plugins control an experiment variable, that is responsible for controlling input variables of executed models. Simulation and Instrumentation plugins are directly used by the experiment layer, as they create the factories for simulation algorithms (`ProcessorFactory`), model instrumenters (`ModelInstrumenterFactory`), and simulation instrumenters (`SimulationInstrumenterFactory`). The plugin for single-run analysis controls a simulation stop policy (respectively the `StopPolicyFactory`), that determines the end time of a simulation run. Similarly, the multi-run analysis plugin controls a replication criterion and the corresponding `ReplicationCriterionFactory`, that determines the number of required replications.

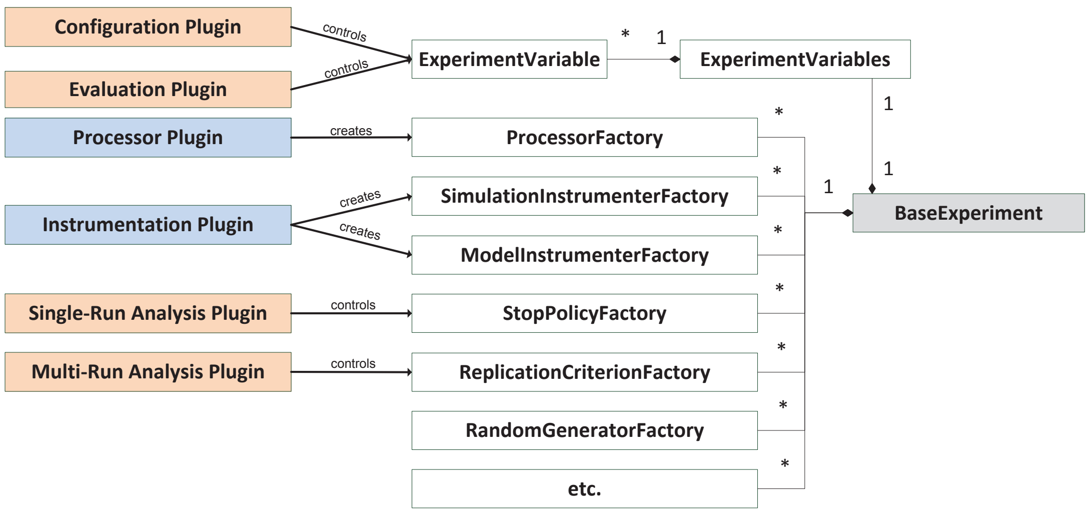


Figure 3.7.: Relation between experiment task plugins and JAMES II experiment layer.

3.3. Experiments Executed with the Extended Experimental Layer

To demonstrate the flexibility and applicability of the experimentation layer extension, four different experiments are presented, that have been realized using it. The experiments deal with the validation of a *Northrup-Allison-McCammon (NAM)* method reimplementation [76], a sensitivity analysis as well as an optimization of a Wnt/β-catenin model [131], and an exploratory analysis of a *Network on Chip (NoC)* model [187]. The code representations of the parameter blocks, i.e., the concrete specifications, for all four experiments are shown in Appendix B.

3.3.1. Validation of a NAM Method Implementation

The first experiment deals with validating a SpacePi-Calculus [105] implementation of the NAM method [140]. SpacePi-Calculus is an extension of the π-Calculus [136] with spatial properties. The validation experiment should prove that the reimplemented NAM method produces similar results as the original one.

3.3.1.1. The NAM Method

The NAM method is one of the most common setups for modeling and simulation of bio-molecular diffusional association [140]. The basic idea behind it is depicted in Figure 3.8. The method investigates the relative movement of two particles in a 3-dimensional space. As the focus is on relative movement, one of the particles (F) is fixed in the center of the coordinate system, whereas the other particle (M) moves freely on the electrostatic potential grid of F. Particles F and M collide (and consequently react) as soon as their distance falls below the *reaction radius* a. Initially, particle M is placed randomly on the *b-surface* — a circle with radius b in the center of the coordinate system. During the execution of the model, the moving path of particle M is calculated. As soon as it moves into the action radius of particle F, the execution is terminated with an end state — the so-called *collision state* — denoting a reaction. Otherwise, if particle M leaves a given radius q — the *q-surface* — the execution is terminated with a different end state — the so-called *exit state* — denoting no reaction.

Figure 3.9 shows the corresponding SpacePi-Calculus model. It consists of three different processes,

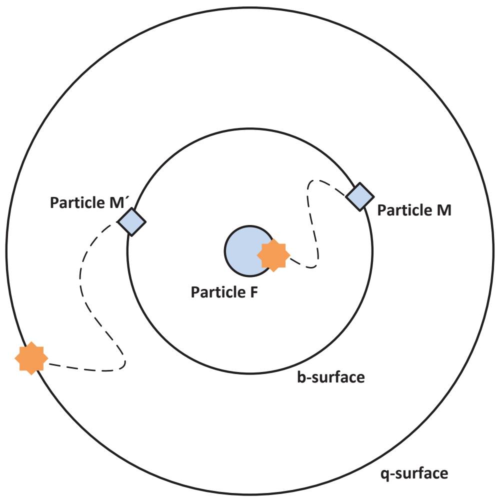


Figure 3.8.: Schematic representation of the NAM method. Initially, the fixed particle (F) is placed in the center, while the moving particle (M or M') is randomly placed on the b-surface, i.e., with a given distance b to the fixed particle F. Particle M associates with particle F at the active site whereas particle M' diffuses into infinite separation (it leaves the q-surface, i.e., its distance to particle F gets high than q)[76].

representing the two particles ($FixedParticle$ and $MovingParticle$) as well as an exit particle for modeling the event that the moving particle escapes (*ExitParticle*). No potential mean force model is specified in this simplified representation. As the relative movement of the particles is modeled, only $MovingParticle$ is associated with the movement function *bMove*. It is initially placed randomly on the b-surface (pos_M), whereas *FixedParticle* is fixed in the center of the coordinate system (pos_F). The action *coll*?$(\sim, a)$ can communicate with the corresponding action *coll*!$(\sim, r)$ of $MovingParticle$ leading to a reaction, which occurs as soon as the two particles reach a distance smaller than the given action radius a. After the collision, i.e., *reaction event*, both processes terminate and the simulation ends.

ExitParticle is placed on the q-surface, and serves as axillary process, to realize an *escape event*. As the SpacePi-Calculus offers no opportunity to represent boundaries, the distance constraint is modeled in the movement function *bMove*. If *FixedParticle* is within the radius q, it updates its position, by adding R to its previous position. R is a ($\mathbb{R}^3$) random variable drawn from a normal distribution with mean 0 and deviation $2 \cdot D\delta_t$, where D is the diffusion coefficient, which has to be provided as model parameter. This corresponds to the mean displacement of a Brownian particle based on the Ermak-McCammon Equation [46]. As soon as *FixedParticle* reaches the q-surface it is placed at the position of *ExitParticle*, leading to a collision between both particles on channel *coll* The mechanic behind this collision is similar to that of *FixedParticle* and $MovingParticle$. Afterward, the processes terminate and the simulation stops in the exit state. Altogether, the model comprises four input parameters: the collision radius a of *FixedParticle*, the radius b of the initial position of *MovingParticle*, the exit radius q, and the *diffusion coefficient D* (determining the ‘speed’ of $MovingParticle$).

The NAM method aims at estimating the rate of diffusional association k, which is computed by:

$$k = k_D(b) \cdot B^\infty(b, q), \tag{3.1}$$

with $k_D(b)$ being the rate constant where two particles reach a certain separation of length b, and B^∞ being the probability that particles react. In the given model, the particles are non-interacting

Position declarations

$pos_F \;:=\; x=0 \;\wedge\; y=0 \;\wedge\; z=0$

$pos_M \;:=\; rand(x,y,z)\ with\ (x^2+y^2+z^2)=b^2$

$pos_E \;:=\; rand(x,y,z)\ with\ (x^2+y^2+z^2)=q^2$

Radius declarations

$a = 10$

$b = 50$

$q = 100$

Potential of mean force declarations

$f_{pmf} : 0$

Motion declarations

$$bMove(pos_t, f_{pmf}) = \begin{cases} pos_{t-1} + R, & if\ pos(x)^2 + pos(y)^2 + pos(z)^2 < q^2 \\ pos_E, & otherwise \end{cases}$$

Process definitions

$FixedParticle = coll?(\sim, a).0$

$MovingParticle[bMove] = coll!(\sim, r).0$

$ExitParticle = coll?(\sim, 0).0$

Initial process

$FixedParticle \mid MovingParticle \mid ExitParticle$

Figure 3.9.: A simple SpacePi-Calculus model based on the NAM model for diffusional association of two particles. An expression $ch?(\sim, r)$ denotes that an empty message is to be received on channel ch with radius r.

spheres, and k_D can be calculated by the Smoluchowski result:

$$k_D(b) = 4\pi D \cdot b, \tag{3.2}$$

B^∞ is usually given by

$$B^\infty(b,q) = \frac{B}{1-(1-B)\Omega} \quad \text{with} \quad \Omega = \frac{k_D(b)}{k_D(q)} \tag{3.3}$$

A valid reimplementation of the NAM method should produce a diffusional association rate k that matches the analytical solution $r_a = 4\pi Da$, where a is the reaction radius (see above). This is the case if the reaction probability (i.e., the probability of two particles to react) B equals the analytical solution:

$$B_{calc} = \frac{a}{b} \cdot \frac{q-b}{q-a} \tag{3.4}$$

Furthermore, B can be estimated by executing a set of simulation runs and calculating the fraction between runs terminating in a collision state and those terminating in an exit state. This estimated probability rate B_{sim} should be close to B_{calc}, and the distance between B_{sim} and B_{calc} denotes the validity of the NAM method.

The NAM Simulator The `NAMSimulator` is integrated into JAMES II as processor plugin and executes the previously described model, by calculating the path of *FixedParticle*. In each time step, it updates the movement and position of particle *FixedParticle* and checks for collisions. Therefore, it uses an RNG and a collision detection algorithm, both of them integrated as plugins as well. Two variants of collision detection have been implemented, a stepwise and an event-triggered version.

In the stepwise version, the distance between *MovingParticle* and the action radius a of *FixedParticle* is calculated at different time steps. A collision is reported if this distance is equal or less than 0. Collisions might happen undetected, if *MovingParticle* touches the collision range in between two time steps. Two types of step-size control have been implemented. The first approach creates a constant step-size over the whole simulation run, which basically ignores the undetected collision problem. The second approach adapts the step-size according to the distance between moving particle and reaction range (the lesser the distance the smaller the time step).

The event-triggered collision detection does not need a step-size control, as it uses the actual movement and position of *FixedParticle* to extrapolate the time point at which the particles should collide beforehand. If no movement updates are scheduled before this time point, the reaction happens. This variant requires more complex calculations, compared to the time-stepped version, but does not miss collisions.

3.3.1.2. The Validation Experiment

Comparing simulation results to analytical solutions is a typical validation experiment [163]. The experiment, presented in the following, shall show the validity of the reimplemented NAM method, by comparing its results to analytical ones. For an overview of the experiment setting see Table 3.1, the concrete specification is shown in Appendix B.1, pp. 129.

Experiment Task	**Realization** *Methods (Selected Methods in Blue)*	*Method Parameters*

Specification	goal: validation; model: NAM model
	configuration: parameter scan; simulation: NAM simulator;
	data collection: final state; single-run analysis: none;
	multi-run analysis: replication estimation; evaluation: visualization	

Configuration	parameter scan ↪`ParameterSetScanModelConfigurator` ↪`ConditionalScanConfigurator`	particle distance $b = 50$ unit lengths collision distance $a = 10$ unit lengths exit distance $q = 100$ unit lengths $D \in \{0.02, 0.2, 2, 20\}$
Model Execution	NAM simulation ↪`NAMSimulator`	collision detection stepwise (fixed step-size) stepwise (adaptive step-size) event triggered random number generator: default Java RNG Mersenne Twister
Data collection	final state ↪`FinalStateObserver`	
Single-run Analysis	-	
Multi-run Analysis	mean estimation ↪`TwoSteppedAnalyzer` ↪`IterativeAnalyzer`	calculate fraction of collision states B_{sim} allowed error $e = 0.05$ confidence $c = 0.95$ underlying distribution: Z_c
Evaluation	visualization ↪`DataStoragePlotter` ↪`MosanEvaluator`	plot B_{sim} in comparison to $B_{calc} = 0.111$ (analytical result)

Table 3.1.: Overview over the steps of a simulation experiment, as well as selected methods and method parameters for the validation of the NAM method [76].

As depicted in Table A.3, p. 128, various methods are available for experimentation in GUISE. For the configuration this includes e.g., parameter search algorithms, different experiment designs, or parameter scan methods. Due to the small parameter space of the investigated model, the `ParameterSetScanModelConfigurator` is used, instead of a more sophisticated parameter search method. It takes a given set of parameters and possible values and creates all possible parameter settings. For the experiment the reaction radius a, b-surface radius b, q-surface radius q, and diffusion coefficient D are considered. In principal b and q can be chosen arbitrarily, as the reaction rate constant is only dependent on the reaction radius a. However, q should be chosen sufficiently larger than b, to allow a significant fraction of replications to react. The present experiment is oriented on the experiment setting proposed by [156], where a has been set to 10 units of length and the fixed distances b and q have been set to 50, respectively, 100 units of length. Furthermore, different diffusion coefficients D have been tried, varying from 0.02 to 20.

The `NAMSimulator` uses components for generating random numbers, for collision detection, and for step-size control if the collision detection works stepwise. Components can be exchanged and extended by exploiting JAMES II's plugin system. Two different RNGs, the default Java RNG and the Mersenne Twister [130] are used, to investigate the impact of changing the RNG component on the simulation results. Furthermore, to measure bias produced by the simulation algorithm, both collision detection variants, i.e., stepwise and event-triggered, are tested as well as both step-size controls, i.e., fixed step-size and adaptive step-size.

Only one observer is available for the NAM method, the `FinalStateObserver`. It monitors whether the two particles have reacted — ending in collision state — or not.

No single-run analysis method is used, because the only important information to calculate B_{sim} is the end state of the simulation, which is provided directly by the observer. Hence, single-run analysis is skipped and the end state is passed through to the multi-run analysis.

For the multi-run analysis the `TwoSteppedAnalyzer`, performing a statistical replication estimation, has been selected. It calculates B_{sim}, by summing up the number of runs ending in a collision state and dividing it by the count of all replications. The number of required replications N is calculated by a two-stage approach [7, pp. 71], which ensures that the final result is inside a given standard deviation e and a given confidence c, according to:

$$N = \left(\frac{Z_c \cdot \sigma_X}{e \cdot \overline{X}} \right)^2 \tag{3.5}$$

with Z_c being the c-quantile of the standard normal distribution, $\overline{X}$ being the mean of the single-run analysis results, σ_X being the corresponding sample standard deviation, and e being the allowed, relative error tolerance. c has been set to 0.95 to achieve a 95 percent confidence interval, with e set to 0.05 leading to 5 percent error tolerance.

To compare B_{sim} to the analytical result B_{calc}, the `DataStoragePlotter` plugin is used. It stores the analysis results in a data storage and offers plotting functions (e.g., line charts). Thereby, a face validation is possible, by comparing B_{sim}, and B_{calc} which has a value of:

$$\tfrac{a}{b} \cdot \tfrac{q-b}{q-a} = \tfrac{10}{50} \cdot \tfrac{100-50}{100-10} = 0.111.$$

If the deviation between B_{calc} and B_{sim} lies inside a given tolerance $e = 0.05$, the model is considered valid.

Experiment Results The results of the experiment are shown in Figure 3.10. As the exchange of the random number generator did not have a significant influence on the results, only the results with the Mersenne Twister are presented. The simulation runs with the event-triggered collision detection generated association rates, that are all inside the defined error tolerance $e = 0.05$. Hence, with this simulator configuration, the reimplemented NAM method produced valid results. However, a significant difference to the analytical results exists, when using the stepwise collision detection. In this case, B_{sim} decreases, as the diffusion coefficient D, i.e., the velocity of the moving particle, is

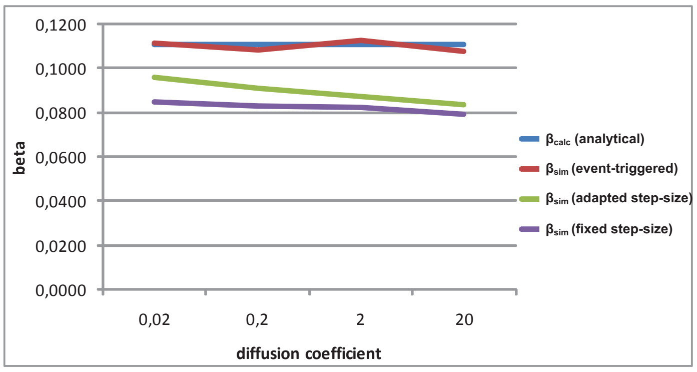


Figure 3.10.: Results of the validation experimen4t.

increased, whereas B_{calc} is constant. The differences between B_{sim} and B_{calc} are higher with the fixed step-size than with the adaptive step-size. An explanation for this behavior lies in the less precision of stepwise collision detection, as collisions happening in between two steps cannot be detected. Hence, if D is low and the particles, therefore, move slowly, the probability of missing a collision is low as well. Similarly, if the diffusion coefficient is high, the probability of missing a collision is high as well. If, on the other hand, the step-size is decreased when the moving particle gets closer to the reaction radius, less — but not all — collisions are missed.

This shows the importance of testing different simulation engines during validation experiments, as the type of collision detection as well as the step-size control had an impact on the experiment results. If, for instance, just the stepwise detection with the fixed step-size would have been used, the biased results would have led to the false conclusion that the NAM model is invalid. Hence, while the experiment shows the validity of the model, it also illustrated the insufficient accuracy of the time-stepped collision detection. This, furthermore, indicates that GUISE can be used for validating simulation engines.

3.3.2. Sensitivity Analysis of a Wnt/β-catenin Pathway Model

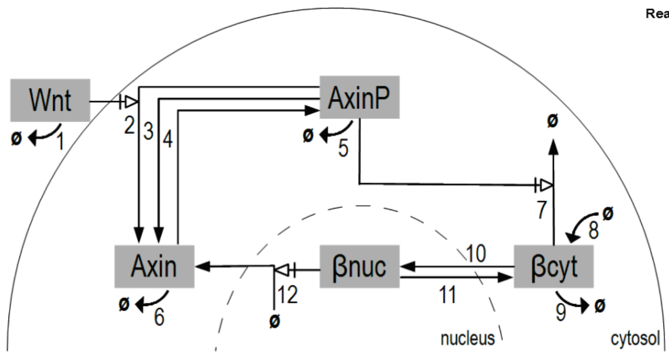
The second experiment is a sensitivity analysis of a Wnt/β-catenin signaling pathway model. The model has been implemented in a discrete-event, species-reaction reimplementation of the model created by Lee [119]. The sensitivity analysis investigates the impact of given reaction rates and species amounts on the behavior of the pathway. Results shall be used to facilitate an optimization experiment that should fit the model to wet-lab experiment data (see Section 3.3.3).

3.3.2.1. The Wnt/β-Catenin Signaling Pathway Model

The Wnt/β-catenin signaling pathway is relevant for the proliferation and differentiation processes of neural cells [28]. Extracellular Wnt molecules induce a reaction cascade within the cell that causes an accumulation of β-catenin in the cytosol. As a consequence, β-catenin moves to the nucleus and activates the transcription of genes. One of those genes is responsible for encoding the Axin protein, which is the major component of a β-catenin destruction complex [127], leading to a negative feedback loop.

The described process has been realized as a species-reaction model that is depicted in Figure 3.11, and comprises five species representing the three main proteins Wnt, β-catenin, and Axin. βcyt and βnuc represent β-catenin in the cytosol and in the nucleus, respectively. $Axin$ and $AxinP$ reflect the phosphorylation state of Axin.

The model, furthermore, includes a set of reaction rules depending on reaction rates. It is, initially, induced with Wnt that decays over time (Rate $k_{\mathrm{W}\downarrow}$, Rule 1), and inhibits the dephosphorylation



Reactions	Rate constant
1	$k_{W\downarrow}$
2	$k_{Ap\Rightarrow A}$
3	$k_{Ap\rightarrow A}$
4	$k_{A\rightarrow Ap}$
5	$k_{Ap\downarrow}$
6	$k_{A\downarrow}$
7	$k_{\beta\Downarrow}$
8	$k_{\beta\uparrow}$
9	$k_{\beta\downarrow}$
10	$k_{\beta in}$
11	$k_{\beta out}$
12	$k_{A\uparrow}$

Figure 3.11.: The Wnt/β-catenin signaling pathway realized as species-reaction model in JAMES II, by Mazemondet et. al. [131].

of *AxinP* (Rate $k_{\mathrm{Ap\Rightarrow A}}$, Rule 2). This influences the phosphorylation (Rate $k_{\mathrm{A\rightarrow Ap}}$, Rule 4) and dephosphorylation (Rate $k_{\mathrm{Ap\rightarrow A}}$, Rule 3) cycle of species *Axin* and *AxinP*. Both species decay over time, as well (Rate $k_{\mathrm{A\downarrow}}$, Rule 6 for *Axin*, and Rate $k_{\mathrm{Ap\downarrow}}$, Rule 5 for *AxinP*). In addition, *AxinP* inhibits the decay of βcyt (Rate $k_{\beta\Downarrow}$, Rule 7). This influences the constant production (Rate $k_{\beta\uparrow}$, Rule 8) and decay (Rate $k_{\beta\downarrow}$, Rule 9) of βcyt that happen simultaneously. β-catenin can move from cytosol (βcyt) to nucleus (βnuc) (Rate $k_{\beta\mathrm{in}}$, Rule 10), and vice versa (Rate $k_{\beta\mathrm{out}}$, Rule 11). βnuc inhibits the production of *Axin* (Rate $k_{\mathrm{A\uparrow}}$, Rule 12), which closes the feedback loop.

The 5 initial species amounts and 12 reaction rates sum up to 17 numerical parameters. The standard values for these parameters correspond to those proposed by Lee [119] and are depicted in Table 3.2. To fit the model to results achieved in the wet-lab, an optimization experiment is conducted and will be described in Section 3.3.3. This section focuses on a sensitivity analysis that is executed for decreasing the relevant part of the parameter space.

Species	Initial amount	Reaction rate	Value
nβcyt	24.9 nM	$k_{\beta\uparrow}$	$0.232\ nmol \cdot L^{-1} \cdot min^{-1}$
nβnuc	24.9 nM	$k_{\mathrm{W\downarrow}}$	$6.65 \cdot 10^{-3}\ min^{-1}$
n*Axin*	0.007 nM	$k_{\mathrm{Ap\Rightarrow A}}$	$7\ min^{-1}$
n*AxinP*	0.042 nM	$k_{\mathrm{Ap\rightarrow A}}$	$0.3\ min^{-1}$
n*Wnt*	100 molecules	$k_{\mathrm{A\rightarrow Ap}}$	$3\ min^{-1}$
		$k_{\mathrm{Ap\downarrow}}$	$0.167\ min^{-1}$
		$k_{\mathrm{A\downarrow}}$	$0.367\ min^{-1}$
		$k_{\beta\Downarrow}$	$4.17 \cdot 10^{-4}\ L \cdot min^{-1} \cdot nmol^{-1}$
		$k_{\beta\downarrow}$	$9.98 \cdot 10^{-5}\ min^{-1}$
		$k_{\beta\mathrm{in}}$	$0.0549\ min^{-1}$
		$k_{\beta\mathrm{out}}$	$0.135\ min^{-1}$
		$k_{\mathrm{A\uparrow}}$	$9.3 \cdot 10^{-5}\ min^{-1}$

Table 3.2.: Standard values of the parameters taken from the model proposed by Lee [119].

3.3.2.2. The Sensitivity Analysis Experiment

The present experiment is conducted to investigate the impact of parameter changes on the model's behavior and identify irrelevant parameters for the subsequent optimization experiment. The experiment description is depicted in Table 3.3, the concrete specification is shown in Appendix B.2.

Different methods have been implemented for GUISE to determine the parameter sensitivity of a model, e.g., parameter scans, or experiment design techniques like Placket and Burman [150]. As the sensitivity of the model with respect to individual parameters is of interest (to facilitate the optimization experiment), the `ConditionalScanConfigurator` plugin has been selected for the configuration

Experiment Task	**Realization**	
	Methods (Selected Methods in Blue)	*Method Parameters*

Specification	goal: sensitivity analysis; model: Wnt/β-catenin signaling pathway model
	configuration: parameter scan; simulation: species-reaction simulator;
	data collection: trajectory of βnuc; single-run analysis: trajectory comparison;
	multi-run analysis: replication estimation; evaluation: sensitivity	

Configuration	parameter scan ↪`ParameterSetScanModelConfigurator` ↪`ConditionalScanConfigurator`	create configurations $I_{p=x}$: increase x (see Table 3.2) until $\mathcal{D}(\mathcal{R}(I_b), \mathcal{R}(I_{p=x})) > 0.1$
Model Execution	species-reaction simulation ↪`FirstReactionMethod` ↪`DirectReactionMethod` ↪`OptimizedDirectReactionMethod` ↪`NextReactionMethod` ↪`TauLeapingMethod`	random number generator: Mersenne Twister
Data collection	trajectory of βnuc ↪`TrajectoryObserver`	trajectory $\vec{y}_i^I$ of βnuc
Single-run Analysis	trajectory comparison ↪`MaximumDistanceTrajectoryComparator` ↪`MinimumDistanceTrajectoryComparator` ↪`AverageDistanceTrajectoryComparator` ↪`SquaredDistanceTrajectoryComparator` ↪`EuclideanDistanceTrajectoryComparator`	distance $O_{I,i}$ between simulation trajectory $\vec{y}_i^I$ and wet-lab trajectory $\vec{y}^w$
Multi-run Analysis	replication estimation ↪`TwoSteppedAnalyzer` ↪`IterativeAnalyzer`	mean $\mathcal{R}$ (I) of outputs $O_{I,i}$ allowed error $e = 0.05$ confidence $c = 0.95$ underlying distribution: Z_c
Evaluation	parameter sensitivity ↪`SensitivityEvaluator`	distance $\mathcal{D}(\mathcal{R}(I_b), \mathcal{R}(I_{p=x}))$ between results of default configuration $\mathcal{R}(I_b)$ and current configuration $\mathcal{R}(I_{p=x})$

Table 3.3.: Overview over the steps of the sensitivity analysis experiment with the Wnt/β-catenin pathway model.

task. It investigates the impact of changing the value of each individual parameter on the simulation results [58]:

$$\mathcal{S}(I_b) = \{(p, min(\mathcal{S}_P(I_b, p))|p \in \mathcal{I}_M\} \tag{3.6}$$

with $\mathcal{I}_M$ being the set of parameters under investigation and I_b being a reference (or default) parameter setting. The reference setting contains the standard values for the parameters, taken form the model proposed by Lee (see Table 3.2). It is applied to $\mathcal{S}_P(I_b, p)$, which performs the concrete scan, i.e., it computes the required relative change of parameter p in order to achieve a relative distance of 0.1 (considered to be significant) in the simulation analysis results:

$$\mathcal{S}_P(I_b, p) = \left\{ \frac{|arg_p(I_b) - arg_p(I_{p=x})|}{arg_p(I_b)} \middle| \frac{|\mathcal{R}(I_b) - \mathcal{R}(I_{p=x})|}{\mathcal{R}(I_b)} > 0.1 \right\} \tag{3.7}$$

where $arg_p(I)$ returns the value of parameter p in parameter setting I, and $\mathcal{R}(I)$ returns the simulation analysis results of I. $I_{p=x}$ is a parameter setting that differs from I_b only in parameter p which gets value x. This value is increased until the desired relative distance of 0.1 is achieved.

For calculating the simulation analysis results, the model is executed by the `OptimizedDirectReactionMethod` plugin which implements an advanced version of Gillespie's direct reaction method [197]. It has been favored instead of other available exact methods (first reaction [67], next reaction [65], or standard direct reaction method [68]), since it has been the fastest *exact* simulation engine in previous experiments with the Wnt/β-catenin signaling pathway model.

As previously stated, the sensitivity analysis is conducted to decrease the parameter space that has to be considered in the subsequent optimization experiment. Hence, the parameters which impact the objective function of the optimization experiment have to be identified. Consequently, the objective value of the sensitivity analysis is the same as that of the optimization (which shall minimize the distance between simulated and wet-lab data). As the focus of the wet-lab experimental observations were the dynamics of nuclear β-catenin (βnuc), the trajectory of βnuc is measured during the observation task.

For the single-run analysis, the simulation trajectory is compared to the wet-lab trajectory. Both trajectories are converted to time series (i.e., comprising equidistant time steps) by interpolation. Out of the set of implemented comparison methods (see Table A.3, pp. 128), the `EuclideanDistanceTrajectoryComparator` has been selected for this task. It implements a standard measure for the similarity between two time series [1], by comparing them according to:

$$O_{I,i} = \delta_{eq}(\vec{y}_i^I, \vec{y}^w), \tag{3.8}$$

where $\vec{y}_i^I = y_{i,0}^I, ..., y_{i,m}^I$ is the time series generated by executing the ith simulation run of parameter setting I, $\vec{y}^w = y_0^w, ..., y_m^w$ is the reference time series achieved during wet-lab experiments, and $\delta_{eq}(\vec{y}_i^I, \vec{y}^w)$ calculates the euclidean distance [41, p. 94]:

$$\delta_{eq}(\vec{y}_i^I, \vec{y}^w) = \sqrt{\sum_{j=0}^{m}(y_{i,j}^I - y_j^w)^2} \tag{3.9}$$

To face randomness in the model, the `TwoSteppedAnalyzer` plugin has been applied. The simulation runs are replicated and the mean of the single-run analysis results for those runs are computed:

$$\mathcal{R}(I) = \sum_{i=1}^{n} \frac{O_{I,k}}{n} \tag{3.10}$$

with $O_{I,i}$ being the simulation result of run i with configuration I. To calculate the required amount of replications, a two stage approach [7, pp. 71] is used to ensure that the standard error is inside a given error tolerance e, with a given confidence c. Similarly to the NAM experiment (see Section 3.3.1), c has been set to 0.95 to achieve a 95 percent confidence interval, and e has been set to 0.05 leading to 5 percent error tolerance.

The calculated mean is used during evaluation by the `SensitivityEvaluator`. It calculates the relative distance between the mean of the currently executed parameter setting $I_{p=x}$ and the reference setting I_b (comprising the standard values of the parameters) according to:

$$\mathcal{D}(\mathcal{R}(I_b), \mathcal{R}(I_{p=x})) = \frac{|\mathcal{R}(I_b) - \mathcal{R}(I_{p=x})|}{\mathcal{R}(I_b)} \tag{3.11}$$

The result is sent as feedback to the configuration task (see Equation 3.7).

Experiment Results The results of the sensitivity analysis are given in Table 3.4.

The model is highly sensitive to changes with respect to βnuc, which is not unexpected, as the βnuc trajectory is a fundamental component of the objective value (distance between wet-lab and simulated data). This is, however, also the reason, why optimizing this parameter is not meaningful.

Axin, however, is a promising candidate for manipulation to change the overall behavior of the model. In both of its states (normal and phosphorylated) it has a crucial impact, as even small changes in the species' amounts lead to significant changes in simulation results. A similar sensitivity has been detected for the production rate ($k_{\beta\uparrow}$) of βcyt, which is surprising, since the rate of decay ($k_{\beta\downarrow}$) has the lowest sensitivity of all parameters. An explanation for this discrepancy might be that an additional decay rate exists ($k_{\beta\Downarrow}$), which is inhibited by $AxinP$ and has a higher influence on the model.

All in all, the parameters in the left column of Table 3.4 have a high impact on the model. A change of less than 10 percent in any of such parameter's values is sufficient to achieve a significant change in the simulation results. These parameters are considered in the optimization experiment.

parameter	change	parameter	change
βnuc	0.0001	$k_{\beta\mathrm{in}}$	0.18
$Axin$	0.016	$k_{\mathrm{A}\to\mathrm{Ap}}$	0.23
$AxinP$	0.018	$k_{\mathrm{W}\downarrow}$	0.25
$k_{\beta\uparrow}$	0.019	$k_{\mathrm{Ap}\Rightarrow\mathrm{A}}$	0.28
Wnt	0.036	$k_{\mathrm{Ap}\downarrow}$	0.37
$k_{\beta\Downarrow}$	0.04	$k_{\beta\mathrm{out}}$	0.46
$k_{\mathrm{Ap}\to\mathrm{A}}$	0.042	$k_{\mathrm{A}\uparrow}$	0.58
$k_{\mathrm{A}\downarrow}$	0.072	$k_{\beta\downarrow}$	0.91
βcyt	0.098		

Table 3.4.: Results of the sensitivity analysis [131] — relative changes of each parameter, that are necessary to achieve a 10 percent (considered a significant) variation in the distance between simulation and wet-lab trajectory. Hence, smaller values mean higher sensitivity.

3.3.3. Optimization of a Wnt/β-catenin Pathway Model

With the set of significant parameters, identified in the sensitivity analysis, an optimization experiment is conducted, to fit simulation results to the data gained in the wet-lab, i.e., to find a model parameter setting that produces a simulation trajectory with minimal distance to the trajectory produced in wet-lab experiments with real cells. The setup is depicted in Table 3.3.

Experiment Task	**Realization**	
	Methods (Selected Methods in Blue)	*Method Parameters*

Specification	goal: optimization; model: Wnt/β-catenin signaling pathway model
	configuration: parameter search; simulation: species-reaction simulator;
	data collection: trajectory of βnuc; single-run analysis: trajectory comparison;
	multi-run analysis: replication estimation; evaluation: parameter search feedback	

Configuration	parameter search ↪HillClimbingModelConfigurator ↪SimulatedAnnealingModelConfigurator ↪TabuSearchModelConfigurator **↪HookeJeevesModelConfigurator** ↪BranchBoundModelConfigurator ↪ParticleSwarmModelConfigurator ↪NSGA2ModelConfigurator	search parameters for: $k_{\beta\mathrm{in}}$, $Axin$, $AxinP$, $k_{\beta\uparrow}$, Wnt, $k_{\beta\Downarrow}$, $k_{\mathrm{Ap}\to\mathrm{A}}$, $k_{\mathrm{A}\downarrow}$, and βcyt to minimize fitness value
Model Execution	species-reaction simulation ↪FirstReactionMethod ↪DirectReactionMethod **↪OptimizedDirectReactionMethod** ↪NextReactionMethod ↪TauLeapingMethod	random number generator: Mersenne Twister
Data collection	trajectory of βnuc **↪TrajectoryObserver**	trajectory $\vec{y}_i^I$ of βnuc
Single-run Analysis	trajectory comparison ↪MaximumDistanceTrajectoryComparator ↪MinimumDistanceTrajectoryComparator ↪AverageDistanceTrajectoryComparator ↪SquaredDistanceTrajectoryComparator **↪EuclideanDistanceTrajectoryComparator**	distance $O_{I,i}$ between simulation trajectory $\vec{y}_i^I$ and wet-lab trajectory $\vec{y}^w$
Multi-run Analysis	replication estimation **↪TwoSteppedAnalyzer** ↪IterativeAnalyzer	mean $\mathcal{R}$ (I) of outputs $O_{I,i}$ allowed error $e = 0.05$ confidence $c = 0.95$ underlying distribution: Z_c
Evaluation	parameter search **↪ParameterSearchEvaluator** ↪ParetoRankingEvaluator	minimize objective: $\mathcal{R}(I)$ cancel after: 2,000 executed configurations

Table 3.5.: Overview over the steps of the optimization experiment with the Wnt/β-catenin pathway model.

The concrete specification is shown in Appendix B.3.

Model execution, data collection, single-run, and multi-run analysis are handled similarly as in the sensitivity analysis experiment, since the objective of the optimization, i.e., the distance between simulated and wet-lab trajectory, are the same in both experiments. Configuration and evaluation realize a parameter search, for which various algorithms exist including tabu search, particle swarm, or simulated annealing (see Table A.1, p. 127). Out of the parameter search algorithms implemented for GUISE, the `HookeJeevesModelConfigurator`, realizing the Hooke and Jeeves algorithm [90], has been selected to execute the configuration task. Corresponding to the configuration, the evaluation method creates feedback by applying the `ParameterSearchEvaluator`. It combines parameter settings and their objective values, i.e., simulation and analysis results, and forwards them as feedback to the configuration method. Furthermore, it checks whether cancel criteria are met. The cancel criterion for this experiment is reached as soon as 2,000 different parameter configurations have been executed.

Experiment Results The results of the optimization experiment are depicted in Table 3.6. It shows the seven best parameter configurations (i.e., those configurations that resulted in a minimal distance between simulated and wet-lab trajectory) after the cancel criteria of 2,000 executed configurations has been met.

	$k_{\beta\Downarrow}$ in $\frac{L}{min \cdot nmol}$	$k_{\beta\mathrm{in}}$ in min^{-1}	Wnt in molecules	$k_{\beta\uparrow}$ in $\frac{nmol}{L^{-1} \cdot min^{-1}}$	$k_{\mathrm{A}\downarrow}$ min^{-1}	$AxinP$ in nM	$k_{\mathrm{Ap}\rightarrow\mathrm{A}}$ in min^{-1}	$Axin$ in nM
1.	$4.23 \cdot 10^{-4}$	0.0476	100	0.173	0.367	0.044	0.3	0.007
2.	$3.0 \cdot 10^{-4}$	0.0515	110	0.111	0.367	0.040	0.1	0.008
3.	$4.52 \cdot 10^{-4}$	0.0498	105	0.154	0.367	0.037	0.4	0.007
4.	$3.85 \cdot 10^{-4}$	0.0488	108	0.154	0.367	0.042	0.2	0.006
5.	$3.17 \cdot 10^{-4}$	0.0472	123	0.123	0.367	0.042	0.2	0.007
6.	$4.14 \cdot 10^{-4}$	0.0542	110	0.128	0.367	0.042	0.5	0.007
7.	$4.19 \cdot 10^{-4}$	0.0531	142	0.107	0.367	0.042	0.1	0.007

Table 3.6.: The seven best parameter configurations after the cancel criteria of 2000 executed configurations.

3.3.4. Exploratory Analysis of a VulcaNoCs Model

The fourth experiment focuses on a NoC model [187, 188] that has been designed to investigate the temperature development in a computer chip and to optimize its temperature management. An exploratory analysis is performed to investigate the model's parameter space.

3.3.4.1. The NoC Model

NoC [18] is a communication paradigm, that was proposed to face scalability issues and synchronization overhead of conventional bus-based systems on computer chips, by replacing them with modern data network connections. Therefore, *Intellectual Property Cores (IPCs)* are linked with distributed routers (one router per IPC) that are connected by short links rather than large global on-chip connections. By using routers, the communication between IPCs is configurable and can be adapted for optimizing e.g., heat distribution, or the overall performance of the chip. Usually, a *Thermal Management Unit (TMU)* is used to reschedule tasks for optimizing the thermal behavior of the NoC, which uses physical sensors to monitor the heat spread and take actions *reactively*. The TMU itself is not a physical component of the NoC but a task that is handled by IPCs.

Tockhorn et. al. [174] introduced a modeling approach for temperature distributions in NoCs, by representing the dynamic thermal behavior of integrated circuits as equivalent electrical *Resistor-Capacitor (RC)* circuits (see Figure 3.12). The overall aim is real-time modeling and predicting the thermal behavior of NoCs, instead of using costly physical sensors. In this case, a *proactive* TMU holds a model of the NoC based on RC tiles to enable those predictions and to react to problematic

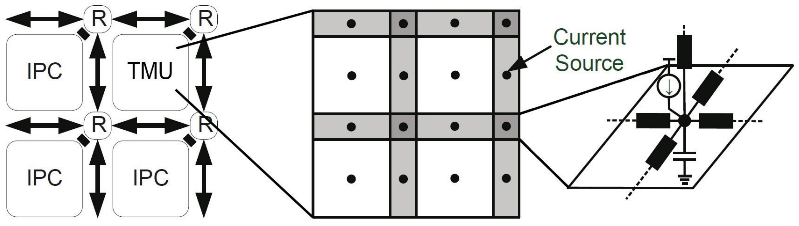


Figure 3.12.: 2×2 NoC with a proactive TMU [187]. The TMU itself resides on an IPC and holds a model of the NoC, represented by 4×4 RC tiles.

thermal states (e.g., by rerouting computational load). In the model, the temperature of a RC tile is represented by the voltage drop over its capacitor. Consequently, the RC circuit's heat flow can be described as current passing through a thermal resistance. This allows a *proactive* working of the TMU, i.e., it does not react to the results of the sensors but precalculates the thermal behavior based on computational load.

For investigating the advantages of this approach, VulcaNoCs, an M&S environment for TMU and RC circuits has been introduced by Wegner et. al. [187], using the SystemC *Analog Mixed Signal (Analog Mixed Signal (AMS))* library [15]. A VulcaNoCs model is stochastic and represents the whole NoC, i.e., the IPC and router layout as well as the TMU including the RC circuit model, which the TMU uses to make decisions. Thus, all three entities, IPC layout, RC-circuit, and TMU, as well as their parameters influence the VulcaNoCs model.

The IPC layout is primarily defined by the number of IPCs —the higher it gets the more complex the model is.

The behavior of the TMU depends on the following parameters:

- T_i —temperature threshold for the components, at which the TMU starts to intervene
- T_a —activity threshold for the components; if it is reached, a signal is sent to the TMU to recalculate the temperature
- B_u —upper bound for the IPC's temperature; if the temperature of an IPC gets higher than this value, tasks may not be rescheduled by the TMU anymore
- B_l —lower bound for the IPC's temperature; if the temperature of an IPC gets lower than this value, tasks may not be rescheduled by the TMU anymore

All four parameters are reflected in the VulcaNoCs model.

The complexity of the RC circuit depends on its resolution for representing the heat flow in the NoC. By altering the number of RC tiles, responsible for modeling one component (IPC, router, or link), different resolutions are possible. So far, four have been realized:

- $BLOCK$ —each component is represented by one RC tile
- $RES1$ —routers are represented by one RC tile, additional components are scaled according to the physical size proportions of the real chip
- $RES2$ —routers are represented by 2×2 RC tiles, additional components are scaled according to the physical size proportions of the real chip
- $RES3$ —routers are represented by 3×3 RC tiles, additional components are scaled according to the physical size proportions of the real chip

Higher resolutions result in more complex and costly but also accurate predictions by the TMU, and in a more complex VulcaNoCs model.

3.3.4.2. The Exploratory Analysis Experiment

The VulcaNoCs model is used to optimize the TMU and underlying RC circuit with respect to temperature, performance (throughput of the computer chip), or a compromise between both. The exploratory analysis should give ideas for further investigations, e.g., optimization experiments, by ranking the parameter settings according to such objectives. Table 3.7 depicts the experiment description, while the concrete specification is presented in Appendix B.4, pp. 133.

Experiment Task	**Realization**	
	Methods (Selected Methods in Blue)	*Method Parameters*
Specification	goal: optimization; model: VulcaNoCs model configuration: parameter scan; simulation: VulcaNoCs simulator; data collection: temperature and performance information; single-run analysis: none; multi-run analysis: replication estimation; evaluation: parameter search feedback	
Configuration	parameter scan	layout: quadratic (2×2 up to 8×8)
	↪`ParameterSetScanModelConfigurator`	$T_i \in \{1.0, 1.5, 2.0, 2.5, 3.0\}$ volts
	↪`ConditionalScanConfigurator`	$B_l \in \{2.0, 3.0\}$ volts
		$T_a \in \{1000, 5000, 10000\}$ operations
		RC circuit: $BLOCK$, $RES1$, and $RES2$
Model Execution	VulcaNoCs simulation	
	↪`VulcaNoCsWrapper`	
Data collection	temperature and performance	
	↪`VulcaNoCsResultObserver`	
Single-run Analysis	-	
Multi-run Analysis	replication estimation	calculate means for all performance data
	↪`TwoSteppedAnalyzer`	allowed error $e = 0.05$
	↪`IterativeAnalyzer`	confidence $c = 0.95$
		underlying distribution: Z_c
Evaluation	parameter search	ranking according to
	↪`ParameterSearchEvaluator`	temperature, performance,
	↪`ParetoRankingEvaluator`	and balanced

Table 3.7.: Overview over the steps of the optimization experiment with the VulcaNoCs model.

To configure the model, the `ParameterSetScanModelConfigurator` (as described in Section 3.3.1, p. 39) is used. The set of generated parameter settings is the cross product of the given parameter values. The experiment is conducted with 7 different quadratic IPC layouts (2×2 up to 8×8). T_i and B_l are given in voltages, that represent temperature thresholds for the modeled RC tiles, whereas T_a is given in executed operations. B_u is kept at its default value 70.0 volts and, therefore, not considered in the parameter settings.

For the resolution of the RC circuit, only the first three variants ($BLOCK$, $RES1$, and $RES2$) are used, as previous experiments showed that $RES3$ is too complex to be calculated. Altogether, $5 \times 2 \times 3 \times 3 = 90$ parameter configurations (see configuration task in Table 3.7) for each of the seven modeled IPC layouts lead to 540 tested configurations.

The model is simulated using the `VulcaNoCsWrapper`, which is a JAMES II simulator plugin wrapping the SystemC AMS library. As the SystemC AMS library is based on the programming language C++, the Java native interface [74] has been used for integrating it. The NoC simulator does not allow an external control of the simulation steps, which hampers the use of a single-run analysis method. A step-wise processing is, however, not necessary, as the NoC simulator collects and aggregates all required output data (retrieved by the `VulcaNoCsResultObserver`) during the run:

1. gross data throughput in bits per clock cycle
2. netto data throughput in bits per clock cycle
3. maximum delay for sending a packet in clock cycles
4. average delay for sending a packet in clock cycles
5. maximum delay for sending a packet header in clock cycles

6. average delay for sending a packet header in clock cycles
7. average delay for routing delay in clock cycles
8. received data packets
9. received data flits
10. average temperature of the NoC in °C
11. maximum temperature of the NoC in °C
12. maximum temperature variation in the NoC in K
13. average temperature variation between adjacent components of the NoC in K
14. deviation between TMU temperature and real temperature
15. number of times IPCs are in unsafe state
16. average time IPCs are in unsafe state
17. maximum temperature of IPCs in unsafe state
18. number of times IPCs are in illegal state
19. average time IPCs are in illegal state
20. maximum temperature of IPCs in illegal state
21. number of times routers are in unsafe state
22. average time routers are in unsafe state
23. maximum temperature of routers in unsafe state
24. number of times routers are in illegal state
25. average time routers are in illegal state
26. maximum temperature of routers in illegal state

As VulcaNoCs comprises stochastic factors, simulation runs have to be replicated and their results aggregated. As in the previous experiments, this is done by the `TwoSteppedAnalyzer`, which calculates the mean results with a confidence c of 0.95 and an allowed error e of 0.05.

The means are used by the `ParetoRankingEvaluator` to generate a pareto ranking of the parameter settings, during the evaluation step. The ranking happens according to one of the following three goals:

- performance: maximize output variables 1,2,8,9 and minimize output variables 3 to 7
- temperature: minimize output variables 10 to 26
- balanced: optimize all output variables (maximize output variables 1,2,8,9, and minimize output variables 3 to 7 and 10 to 26)

As a pareto ranking is performed, parameter settings with a higher count of optimal objective values get a higher rank than those with fewer optimal values. Hence, the highest ranked parameter setting has the highest count of optimal objective values.

Experiment Results Table 3.8 shows the top ten parameter settings of the performance, temperature and balanced ranking. Only the results of the 2×2 NoC are depicted exemplarily for the experiments with other NoC resolutions that showed similar results. The best, i.e., highest-ranked, settings can be used as blueprint for the RC circuit and TMU configuration to manage a NoC.

The performance ranking seems to depend highly on T_i, i.e., the point at which the TMU starts to intervene. It is low in all top ten configurations, varying between 1.0 and 1.5. This implies, that a very active TMU is beneficial for the computational power of the NoC.

The temperature ranking on the other hand, also includes higher values for T_i in its top ten parameter settings. In the first eight settings T_i varies between 2.0 and 2.5, which leads to the conclusion, that a faster reaction of the TMU on rising temperatures does not necessarily improve the overall thermal behavior. The resolution of the RC circuit, however, seems to have influence on the thermal behavior, as all top ten settings use the $RES2$, i.e., the most detailed, circuit resolution. Hence, a higher resolution leads to more reliable TMU predictions for the temperature and, consequently, to more efficient reactions.

For the balanced ranking, the RC circuit resolution is important as well, as $RES2$ is used in all top ten parameter settings. In addition, B_l has an impact — nine out of the ten settings include a value of

	Rank	Resolution	T_i	T_a	B_l
performance ranking	1.	$RES2$	1.5	10000	2.0
	2.	$RES1$	1.0	1000	2.0
	3.	$RES1$	1.0	5000	3.0
	4.	$RES1$	1.0	1000	3.0
	5.	$RES2$	1.0	1000	2.0
	6.	$RES2$	1.0	5000	3.0
	7.	$RES2$	1.0	10000	2.0
	8.	$RES2$	1.5	10000	3.0
	9.	$RES2$	1.0	10000	3.0
	10.	$RES2$	1.5	5000	3.0
temperature ranking	1.	$RES2$	2.5	10000	3.0
	2.	$RES2$	2.0	10000	3.0
	3.	$RES2$	2.0	5000	2.0
	4.	$RES2$	2.5	5000	3.0
	5.	$RES2$	2.5	10000	2.0
	6.	$RES2$	2.0	5000	3.0
	7.	$RES2$	2.5	1000	3.0
	8.	$RES2$	2.5	1000	2.0
	9.	$RES2$	1.0	10000	3.0
	10.	$RES2$	1.0	10000	2.0
compromise ranking	1.	$RES2$	2.5	10000	3.0
	2.	$RES2$	1.0	5000	3.0
	3.	$RES2$	1.0	10000	3.0
	4.	$RES2$	2.0	10000	3.0
	5.	$RES2$	2.0	5000	3.0
	6.	$RES2$	2.5	5000	3.0
	7.	$RES2$	1.5	10000	3.0
	8.	$RES2$	1.5	5000	3.0
	9.	$RES2$	1.0	10000	2.0
	10.	$RES2$	1.0	1000	3.0

Table 3.8.: The first 10 configurations of rankings according to performance, temperature, and compromise, for a 2×2 NoC.

3.0 for this parameters, i.e., the lower bound for the IPC's temperature at which tasks are rescheduled is high. Hence, for gaining a compromise between performance and temperature behavior, a TMU that waits for a higher temperature level before becoming active, is beneficial.

3.4. Summary

This chapter extended the JAMES II experiment layer in order to support the six tasks of a simulation experiment, identified in Chapter 2. Therefore, the tasks have been arranged in a layered structure (managing the data flow) that communicates with existing components of JAMES II experiment layer (Section 3.2). For realizing tasks, the plugin mechanism of JAMES II has been exploited, to integrate and exchange different methods for each task (Section 3.1). Four example experiments from three different application areas (molecular biology, cell biology, and electrical engineering), and pursuing four different goals (validation, sensitivity analysis, optimization, and exploratory analysis), have been conducted to show the flexibility and suitability of the experiment structure for different experiment types (Section 3.3). However, additional features are required for proper guidance through an experiment.

The first refers to experiment description and documentation. Experiments have to be described by parameter blocks, which is a very flexible but not very user-friendly approach. In addition, no means for *documentation* and *data provenance* (i.e., explicit data documentation throughout the experiment) are provided. Both features are, however, mandatory to achieve repeatability, a key requirement for experimental science.

Experiment description and documentation can be integrated, by mapping the experiment structure to workflows. Workflows can be described with workflow representation languages (e.g., workflow nets [179], or Business Process Execution Language (BPEL) in [107]) that include well-defined semantics, and can be used for a more convenient description of experiments. Monitoring mechanics, inherent to workflow management systems, can be used to document the experiment. These features, in combination with the high flexibility of workflow descriptions, make workflows the ideal base for experiment execution in GUISE. In addition, a realization as workflow reflects common standards, since well defined workflows are considered to be essential for repeatable processes [100]. The use of workflows increases quality and credibility of experiments, by offering a stepwise track of the experiment process, intuitive experiment description, auto documentation, and progress information. The realization of the GUISE experiment workflow will be described in Chapter 4.

A second feature, relevant for user guidance that is not realized in the experimentation layer, so far, is the selection of methods to be used (in particular analysis and parameter configuration methods). All of the described experiments have been conducted with the support of at least one domain expert. The experiment with the NAM method could only be conducted after a specific selection of model parameter values for the parameter scan, and by testing the different available simulation engine components. The experiments with the Wnt/β-catenin Pathway Model (Section 3.3.2) needed assistance to find the right configuration and single-run analysis method. For both tasks various methods exist to achieve the desired goal, e.g., for single-run analysis, methods based on calculating the maximum average, or squared distance could have been used instead of the Euclidean distance method. Finally, the NoC experiment needed assistance in choosing the parameter scan method, and the ranking algorithm.

As an alternative to taking advantage of a domain expert, the decision between algorithms could be automated by selection & composition mechanisms that consider problem and algorithm features, e.g., features of the trajectories when selecting a method for single-run analysis. To allow such a selection, an infrastructure is required that allows to identify features and learn which method is the most suitable or how methods can be orchestrated to solve a specific problem. The realization of the GUISE selection & composition mechanism will be described in Chapter 5.

4. Mapping the Experiment Structure to Workflows

"I long to accomplish a great and noble task, but it is my chief duty to accomplish small tasks as if they were great and noble."

Helen Keller

In the previous chapter, the experimentation layer of JAMES II has been extended for an explicit and flexible support of the six experiment tasks identified in Chapter 2. The corresponding experiment structure shall be mapped to workflows to enrich it with the experiment description and documentation features provided by a workflow management system.

A variety of such systems exist, e.g., KEPLER [3], a very general tool for defining — and hence guiding — workflows of scientific processes. It exploits the PTOLEMY II [177] framework for experimentation with actor-oriented models, by using such models to represent workflows for processing scientific data, like streaming sensor data, medical and satellite images, simulation output, or observational data. As a multi-purpose tool for generating scientific workflows, KEPLER is comparable to the WORMS framework [159], which has been designed to generate workflows in the context of modeling and simulation. Although, neither KEPLER nor WORMS directly provide a concrete workflow for guiding users through a simulation experiment, both tools can be used for designing it. WORMS is of special interest in this case, as it is built for M&S, and, hence, will be used for creating the workflow realization of the previously designed experiment structure.

This chapter will present and discuss this workflow realization, and is divided into three sections. The first section subsumes main features and concepts that ship with WORMS, while the second one derives the concrete experiment workflow. The final section presents workflow realizations of the example experiments introduced in Section 3.3, pp. 36, to show the workflow's applicability.

4.1. Overview of the WorMS Workflow Management System

WORMS is a workflow support framework for M&S, developed by Rybacki et. al. [159]. It is based on the concepts of business process modeling and scientific workflows, and uses a plugin structure conforming to the strategy pattern [63] by offering extension points where custom 'strategies' (i.e., components) can be plugged in. In addition, fixed components exist that are responsible for managing and synchronizing the plugins. For representing workflows, WORMS relies on workflow nets [181] as intermediate representation allowing verification and analysis of workflows, including the creation, redefinition, and improvement of them.

4.1.1. Components

The components of WORMS are well-separated parts illustrated in Figure 4.1 and explained as follows.

The central component for executing a workflow is the *Workflow Engine*. It translates workflow models into the intermediate format by using a *Converter* and triggers a possible analysis of the model provided using an *Analysis* component. Different representations, like XML Process Definition

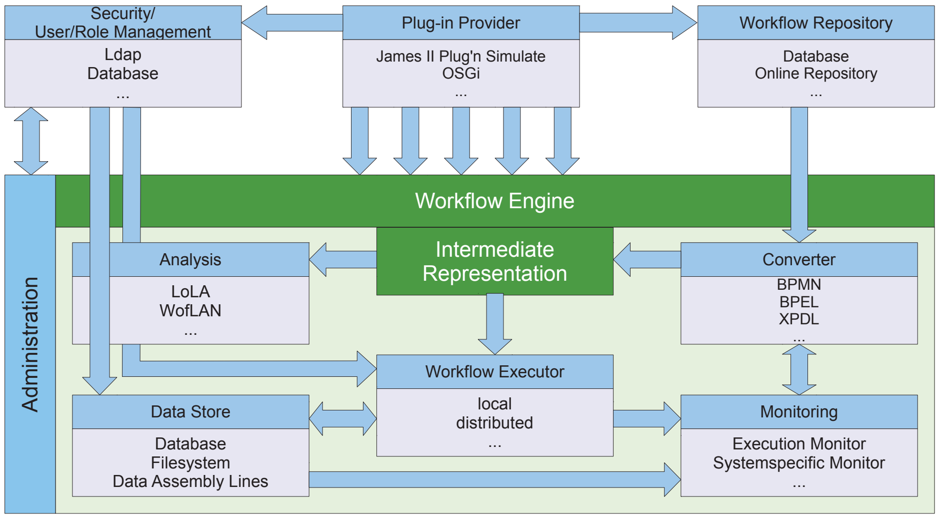


Figure 4.1.: Components of the WorMS framework and their relations, by Rybacki et. al. [159].

Language (XPDL) [32] or BPEL [106] can be processed by WorMS as long as an according *Converter* is present. The *Analysis* component allows the integration of different analysis algorithms and external tools, e.g., LoLA [165] or WofLAN [183] for model-checking Petri nets. This is especially beneficial for debugging purposes.

After the conversion, the workflow model is instantiated and passed to a *Workflow Executor*, which is exchangeable to support different execution strategies. This is beneficial to achieve the *collaborations & delegations* and *detached execution* requirement for scientific workflows [66], as *Workflow Executor* components can be implemented that allow, e.g., the execution on different platforms, the transfer of a running workflow between different machines, or the detached execution of a workflow or task. After being implemented, all of these execution schemes can be applied to arbitrary workflow models.

Security and User/Role Management implementations are responsible for authentication, authorization and user/role management to ensure the security of workflow execution, e.g., by restricting workflow execution to a specific user group. An exchange of the *Security and User/Role Management* component is as well possible to support different strategies, e.g., security levels. In addition, implementations might provide further extension points, like the integration of different user management, authentication, or authorization elements.

The data-flow from one task to another is done by a *Data Store* that, furthermore, holds information about the execution (e.g., durations of tasks) to allow repetitions of workflow executions and documentation. This component can be exchanged to support various data storage systems, e.g., file-based, memory-based, or database. To maintain security, the *Data Store* restricts information access with respect to the specific task and user/role at hand (determined by the *Security and User/Role Management* component).

The *Data Store* works together with the *Monitoring* component to realize logging and documentation. *Monitoring* is central for documentation, provenance and reproducibility of workflow runs, as it tracks what, when, where, and how things are done during workflow execution. In additions, it retrieves the used system, machine, and software components to achieve provenance and reproducibility. By using a monitoring component specifically tailored for an M&S system, more specific information, such as model and simulation configurations, events, or used simulator components, can be recorded. Based on the collected information, workflows can be checked, reengineered, profiled, and improved.

The *Administration* component is responsible for control and administration of workflow execution, including, e.g., starting, pausing, stopping, and restarting a workflow. Furthermore, it manages security, by adding or removing users, mapping them to roles, changing privileges, etc.

To support the development of workflows, the *Workflow Repository* holds all available workflow models as well as meta-data, like version history, descriptions, or productivity indicators (valid, broken, analyzed, etc.). This adds to provenance and reproducibility, as access to used workflows is possible even if newer and improved versions are available.

Similar to JAMES II, WorMS uses plugins to integrate and exchange components. However, the concrete plugin mechanism is also exchangeable, as the *Plugin Provider* realizes an abstract plugin mechanism where concrete plugin systems, e.g., the registry of JAMES II (see Section 3.1, pp. 29), can be integrated. This keeps the framework extensible and flexible within different M&S systems. While the *Plugin Provider* interacts with the external mechanism to provide plugins that are not part of WorMS, an internal plugin mechanism is used to handle plugins provided by WorMS itself (e.g., standard converters, executors and datastores). Furthermore, it works as fallback in case an M&S system does not provide a plugin mechanism.

4.1.2. Frames and Templates

WorMS exploits the *templates and frames* paradigm [22] to facilitate the creation and redefinition of dedicated workflows [161], By using an according *Plugin Provider*, templates and frames can be directly mapped to the plugin types (templates) and plugins (frames) of JAMES II [81], Similarly to plugin types, templates can be used for integrating new functionality. Concrete realizations of such functionality are realized as frames.

Usually, many alternatives exist for filling a template with a concrete frame. It can be filled before executing the workflow (allowing the integration of custom frames), or it can be filled on demand during the execution to integrate the most suitable alternative in the current situation. Both approaches can also be mixed, i.e., only a subset of template activities is filled before the execution while the remaining are filled on demand.

The on demand filling happens in two stages:

Filtering During the first step, alternatives are filtered according to the used template, i.e., only those alternatives can be used that correspond to the template, e.g., if the template demands a simulation run execution, only frames realizing a simulation run are applicable. Further filter criteria can be applied, which may depend on information created during runtime, e.g., results of previous activities. Therefore, each method used in the workflow implements the `IFrame` interface, providing the type of information that is required to execute the method in the appropriate context. For instance, a steady state estimator, that uses a time series as input, communicates the need for such information in order to be integrated properly as a frame in the workflow.

Selection During the second step, the frame that shall be actually used is selected from the set of filtered frames. In the default case, this selection happens randomly. Alternatively, the user decides which of the available activities or sub-workflows is used. For supporting users in their decisions, selection can also happen automatically, according to given criteria. In the ideal case, it is sufficient to define such criteria and a manual selection is not necessary. The type of criteria can vary depending on the selection mechanism. Examples for criteria are performance, accuracy of the desired results, or the sub-goal that has to be fulfilled at the specific task.

4.1.3. Intermediate Workflow Representation

The way of modeling and representing workflows is a major question, for which multiple approaches exist, including XPDL [32], BPEL [106], Business Process Model and Notation (BPMN) [189], or Petri net extensions like Workflow Nets [179] and Information Control Nets (ICNs) [44].

WorMS uses workflow nets [179] as intermediate workflow representation. Such representations are based on Petri nets [182, 180], a modeling formalism comprising well-defined semantics allowing various analysis techniques, e.g., deadlock detection [139, 165, 102]. Working on an intermediate representation facilitates the seamless interaction between the WorMS components and allows performing different techniques on the same workflow implementation, like execution, analysis, and conversion.

Despite the use of workflow nets as intermediate representation, modeling workflows is not restricted to this formalism. As mentioned before, converters can be used to transform another workflow representation into a workflow net (e.g., as shown for BPEL by Hinz et. al. [107]) and benefit from the corresponding tools.

4.1.4. Documentation

An important feature of WorMS is the opportunity to document workflow execution and, therefore, the simulation experiment. This includes logging executed steps (e.g., the six tasks of a simulation experiment, see Section 2.3.1, pp. 25), used algorithms, environment, and internal selection made by the software, e.g., partitioning for distributed executions. The *Monitoring* component of WorMS is responsible for this documentation, by providing concrete functionality for tracking the workflow execution, software components, and the machine where the workflow is executed. For instance, for the interaction with JAMES II a logging mechanism is applied, that monitors the registry and keeps track of the used plugins for tracing executed activities. The used hardware and software environment are retrieved by standard Java methods [75]. All this information is stored in the *Data Store* component for further processing or evaluation.

4.2. Representing the Experiment Structure as Workflow

In the following, the workflow mapping of the experiment structure described in Chapter 3, by using WorMS, is explained. The experiment structure is translated into a workflow net, i.e., WorMS' intermediate format to represent workflows. Tasks are represented as workflow net transitions, whereas places and their markings represent the data that are exchanged between them. As the resulting workflow is oriented on the layered view of a simulation experiment (see Section 3.2, pp. 34), it is described firstly on the topmost (experiment) layer and refined in the following. The four layers are reflected by four workflows, or respectively sub-workflows residing in frames that can be exchanged and reused. Thereby, different experiment types can be realized in different contexts, if necessary.

As the goal of the workflow implementation is to cover a broad range of experiments, specific workflow activities are declared as template activities, i.e., they are exchangeable. They can be exchanged by other activities or sub-workflows, collected in frames. For realizing experiment methods, the plugin types presented in Section 3.1.2, pp. 31 are reused. Hence, concrete algorithms are integrated as plugins, allowing a simple integration and exchange of them.

For understanding the workflow figures, note that template transitions are marked with an arrow (denoting that they have to be refined further), and labels on edges represent conditions, i.e., a labeled edge can only be passed by a token, if the according condition is true.

4.2.1. Experiment Workflow

The primary workflow triggers the experiment (see Figure 4.2) and conforms to the experiment layer introduced in Section 3.2, pp. 34.

It starts with a token at the starting place, containing the experiment specification in form of a parameter block that includes, e.g., the *Uniform Resource Identifier (URI)* [19] of the used model. The URI is a common standard for specifying the location of a resource, making it accessible locally as well as remotely, e.g., through the Internet. The token is consumed by the `ExperimentSetupTransition` that generates some initial information, like the experiment ID, which is, for instance, required by data storages. Such information together with the specification is forwarded as token to the `MultiConfigurationProcessingTemplate` that executes the experiment. After the experiment is

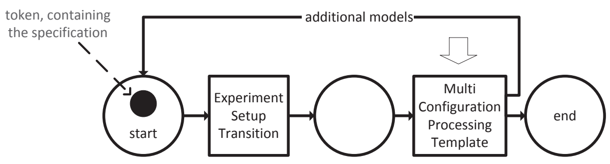


Figure 4.2.: Workflow net for processing the experiment.

terminated, the experiment setup can adapt the specification (by using, e.g., an evolutionary algorithm that considers the experiment results as objective of the specification) and trigger a new experiment execution. This allows an exploration of the specification space (see Figure 3.5, p. 34).

4.2.2. Multi-Configuration Workflow

Figure 4.3 shows the frame for the `MultiConfigurationProcessingTemplate`, that realizes the execution of the given model with multiple parameter settings. Model and simulator parameter settings

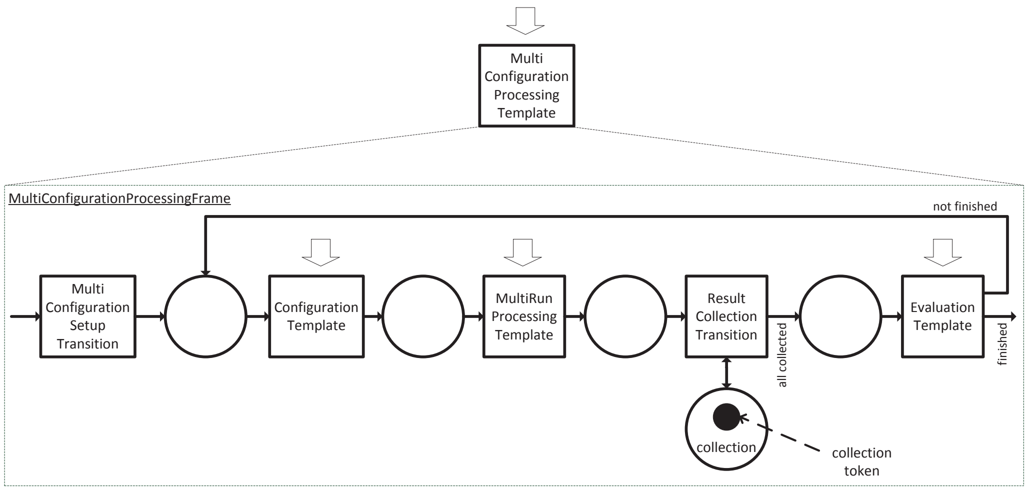


Figure 4.3.: Workflow net for processing the multi-configuration level.

are generated by the `ConfigurationTemplate`. According to the experiment specification, it generates a set of tokens, each containing a model and simulator parameter setting to be executed. The `MultiRunProcessingTemplate` consumes these tokens and executes a set of simulation runs for each of them, using the corresponding model and simulator setting. Different tokens can be executed in parallel, depending on the underlying hardware architecture, e.g., a network, a multi-core machine, etc.

The results of the executions are forwarded to the `ResultCollectionTransition`, which collects results until all tokens have been processed. A special *collection token*, holding the results of processed parameter settings, is used and exchanged between this transition and a collection place, to realize the collection mechanism. To fire, the transition requires a token from the multi-run processing, containing simulation results, and the collection token. The simulation results are added to the collection token which is returned to the collection place afterward. Hence, as the parameter setting tokens are processed, their simulation results are stored.

After all parameter settings have been processed (and all data have been collected), the collection token, including all simulation results, is moved to the `EvaluationTemplate` that is responsible for creating results and feedback. Feedback (e.g., objective values) is forwarded back to the

`ConfigurationTemplate`, starting the next iteration of generating parameter settings, executing and evaluating them. Workflows for configuration and evaluation of parameter settings are depicted in Figure 4.4 and Figure 4.5.

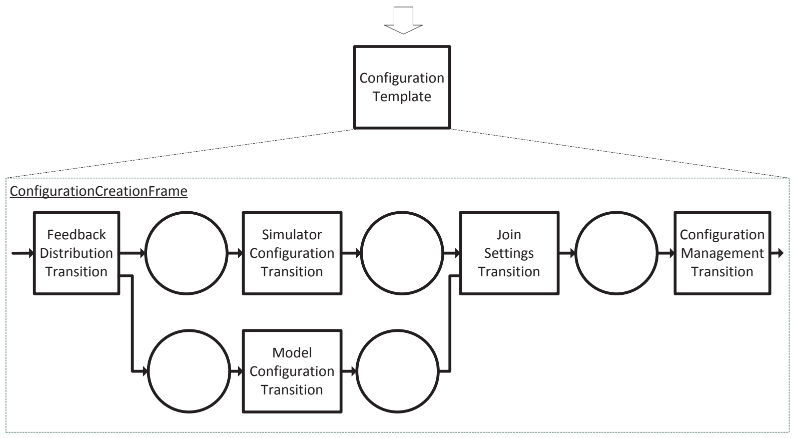

Configuration Template
ConfigurationCreationFrame
Feedback Distribution Transition
Simulator Configuration Transition
Join Settings Transition
Configuration Management Transition
Model Configuration Transition

Figure 4.4.: Workflow net for creating model and simulator configurations.

For the generation of parameter settings, the `FeedbackDistributionTransition` transition distributes possible feedback that might have been generated during evaluation, between `ModelConfigurationTransition` and `SimulatorConfigurationTransition`. Both transitions use configuration algorithms integrated as plugins and generate parameter settings (for model and simulator) that are combined by the `JoinSettingsTransition`, which associates each model parameter setting with each simulator parameter setting, i.e., n model parameter configurations and m simulator parameter configurations, lead to $n \times m$ combinations. For each combination, the `ConfigurationManagementTransition` creates a token to be processed as output of the frame. Note that an alternative strategy of generating parameter settings can be easily realized and integrated, by implementing a different frame for this purpose.

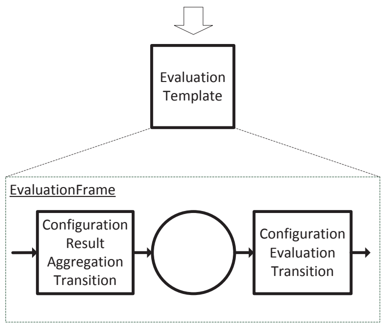

Evaluation Template
EvaluationFrame
Configuration Result Aggregation Transition
Configuration Evaluation Transition

Figure 4.5.: Workflow net for evaluating the results of (model and simulator) configurations.

The same is true for the evaluation strategy, which, in its current implementation, comprises two transitions. The `ResultAggregationTransition` is responsible for generating feedback from the collected simulation results, that is used by the `EvaluationTransition` for creating evaluation results. Both, feedback and results, are returned as output of the frame.

In addition to designing alternative frames, the multi-configuration workflow is adaptable by exchanging the model and simulator configuration algorithms — realizing the configuration step, as well as result aggregation and evaluation algorithms — realizing the evaluation step. All of these algorithms are integrated as plugins.

4.2.3. Multi-Run Workflow

Figure 4.6 depicts the frame for the `MultiRunProcessingTemplate` which executes a set of simulation runs with the same model and simulator parameter setting.

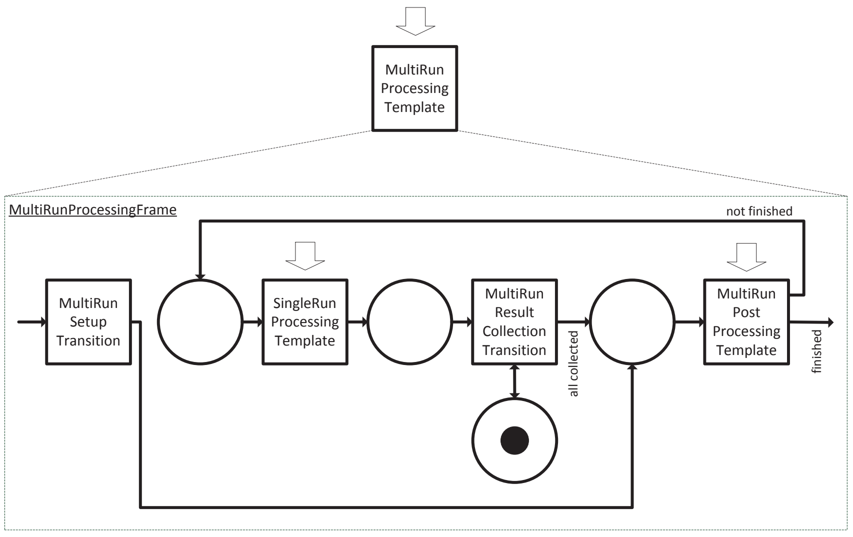

MultiRun Processing Template
MultiRunProcessingFrame
not finished
MultiRun Setup Transition
SingleRun Processing Template
MultiRun Result Collection Transition
all collected
MultiRun Post Processing Template
finished

Figure 4.6.: Workflow net for processing a set of single simulation replications.

The frame starts with a setup transition that produces the required information for running the simulation replications, and triggers the `MultiRunPostProcessingTemplate` afterward. This transition is, primarily, responsible for analyzing the results of executed simulation runs, but also estimates the number of required replications and, hence, has to be called at the beginning to give an initial estimate on the replication number. Based on the estimate, the according replications are executed

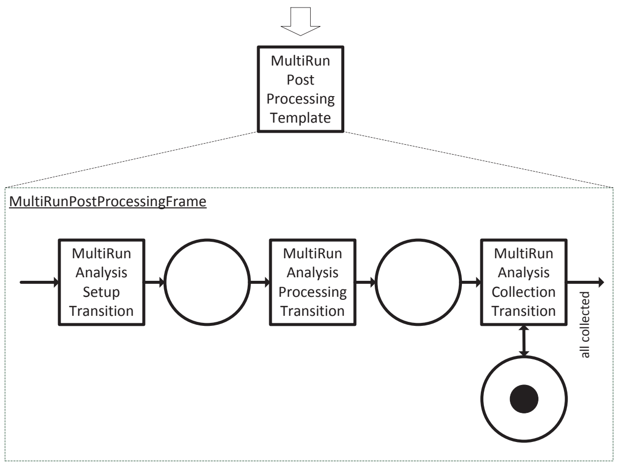

MultiRun Post Processing Template
MultiRunPostProcessingFrame
MultiRun Analysis Setup Transition
MultiRun Analysis Processing Transition
MultiRun Analysis Collection Transition
all collected

Figure 4.7.: Workflow net for typical post processing of a set of single simulation replications.

in the `SingleRunProcessingTemplate` that realizes the next level of the experiment. Results are collected by the `MultiRunResultCollectionTransition`, which works similarly to the result collection mechanism described in Section 4.2.2. As soon as the estimated amount of replications has been executed, results are forwarded to post processing, for analysis and giving a new estimate on the number of required replications. If all required replications have been executed (checked by the corresponding condition) the workflow is terminated and analysis results are returned. Otherwise, additionally required replications are executed.

Figure 4.7 shows the frame realizing the post processing. A setup transition starts the analysis process by creating as many tokens as analysis algorithms shall be executed — multiple algorithms might be executed to support, e.g., multi-objective parameter search. Each token contains the parameters for creating the algorithm as well as simulation run results that came as input into the frame. The tokens are processed (i.e., the analysis executed) by the `MultiRunAnalysisProcessTransition`. The analysis algorithm is integrated as plugin (based on the plugin type described in Section 3.1, pp. 29). Analysis results are, finally, collected and returned by the frame.

4.2.4. Single-Run Workflow

The single-run workflow executes single simulation runs, and implements the lowest level of the experiment as frame for the `SingleRunProcessingTemplate` (see Figure 4.8). A setup transition prepares

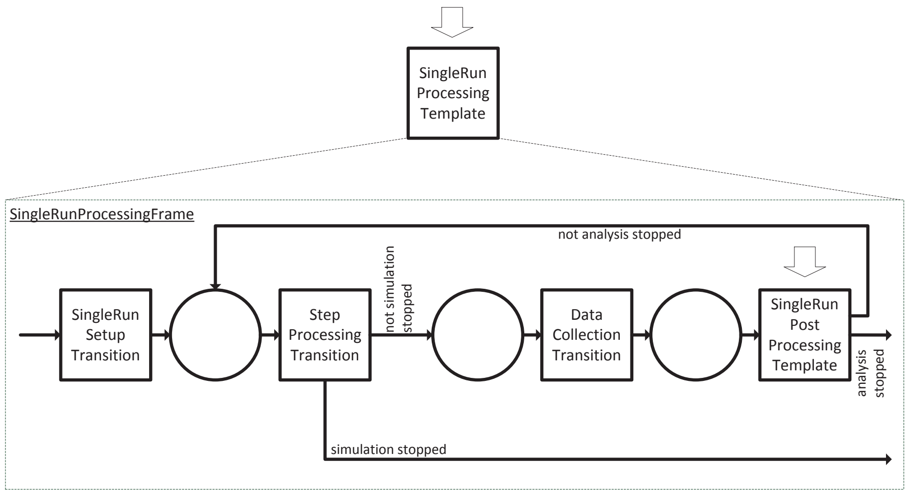

SingleRun Processing Template
SingleRunProcessingFrame
not analysis stopped
SingleRun Setup Transition
Step Processing Transition
not simulation stopped
Data Collection Transition
SingleRun Post Processing Template
analysis stopped
simulation stopped

Figure 4.8.: Workflow net for processing a single simulation run.

the required information for this workflow (e.g., parameters for simulator and analysis components) and initializes the model with the given parameter setting. The prepared information is forwarded to the `StepProcessingTransition` that executes one step of a simulation run. To do that, the transition uses a simulation algorithm that is integrated as plugin of the processor plugin type described in Section 3.1, pp. 29, and instantiated with the given simulator parameter setting. After each step, data are collected by the `DataCollectionTransition` and forwarded to the `StepPostProcessingTemplate`, which is responsible for analysis. Afterward, if additional steps have to be processed (since the analysis needs additional data), an additional simulation step is executed. The workflow can be terminated at two points, either after executing a simulation step — if the simulation itself is finished (i.e., the model reached a final state), or after the post processing — if the analysis is finished (i.e., a reliable analysis result could be obtained).

Figure 4.9 shows an implemented frame for the post processing, comprising four transi-

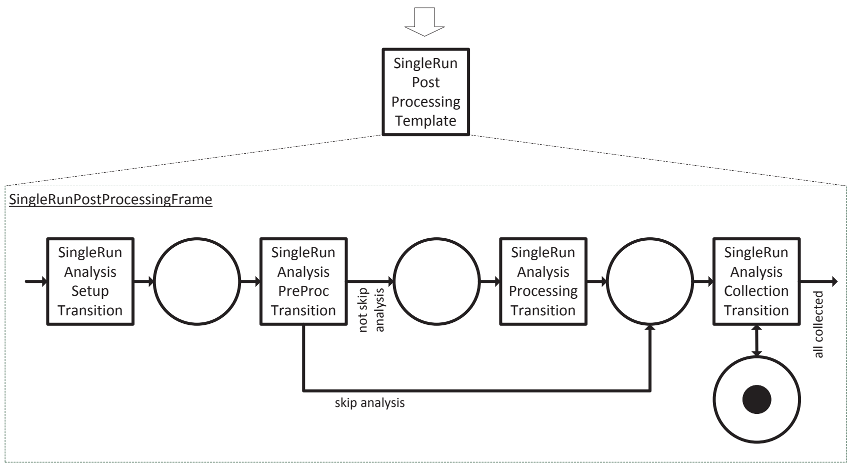

SingleRun Post Processing Template
SingleRunPostProcessingFrame
SingleRun Analysis Setup Transition
SingleRun Analysis PreProc Transition
not skip analysis
SingleRun Analysis Processing Transition
SingleRun Analysis Collection Transition
all collected
skip analysis

Figure 4.9.: Workflow net for typical post processing of a simulation step.

tions. Similar to the multi-run post processing (Section 4.2.3), the first transition creates as many tokens as analysis algorithms shall be executed. These tokens are processed by the `SingleRunAnalysisPreProcTransition` which prepares data for the following analysis. This is necessary, e.g., for realizing an interpolation of simulation data to create a time series from simulation output. Furthermore, the transition determines whether the following analysis should be skipped for the current simulation step, or not. This allows a data analysis in chunks (e.g., each 100 steps) to avoid triggering analysis in each simulation step and saving computational resources. If the analysis is not skipped, it is executed by the `SingleRunAnalysisProcessTransition`. The results are collected and, after all analysis methods have been executed, returned by the frame. Pre-processing and analysis methods are integrated as plugins.

4.3. Using the Workflow to Realize Experiments

To illustrate the applicability of the presented workflow for simulation experiments, this section shows how the experiments described in Section 3.3, pp. 36, can be realized with its help. To avoid redundancy, only those sub-workflows are discussed that directly depend on plugins for executing transitions, since the behavior of the remaining transitions is similar in each experiment and has already been described in the previous section. For instance, the multi-run (sub-)workflow is not discussed again, as its transitions only deal with triggering simulation replications and their analysis, while the concrete handling of both tasks is done in its sub-workflows.

4.3.1. Validation of a NAM Method Implementation

Figure 4.10 shows the relevant sub-workflows for executing the validation experiment of the NAM method implementation, presented in Section 3.3.1, pp. 36. The `ParameterSetScanConfigurator` generates model parameters as depicted in Table 3.1, p. 39, whereas the `ProcessorComponentConfigurator` generates parameters for the simulation engine, e.g., including the used RNG or collision detection.

A default method is used for result aggregation, which sorts results according to the parameter settings that have been used to create them. During evaluation, the `DataStoragePlotter` stores and plots the sorted results, as in the original experiment (also shown in Table 3.1, p. 39).

On the multi-run level, analysis is done by the `TwoSteppedAnalyzer`. On the single-run level, the `NAMSimulator` executes simulation steps, while `FinalStateObserver` retrieves the final state of a simulation run. A single-run analysis is not necessary in this experiment, as only the final state of each simulation run is of interest. Hence, default methods are used that pass collected data through.

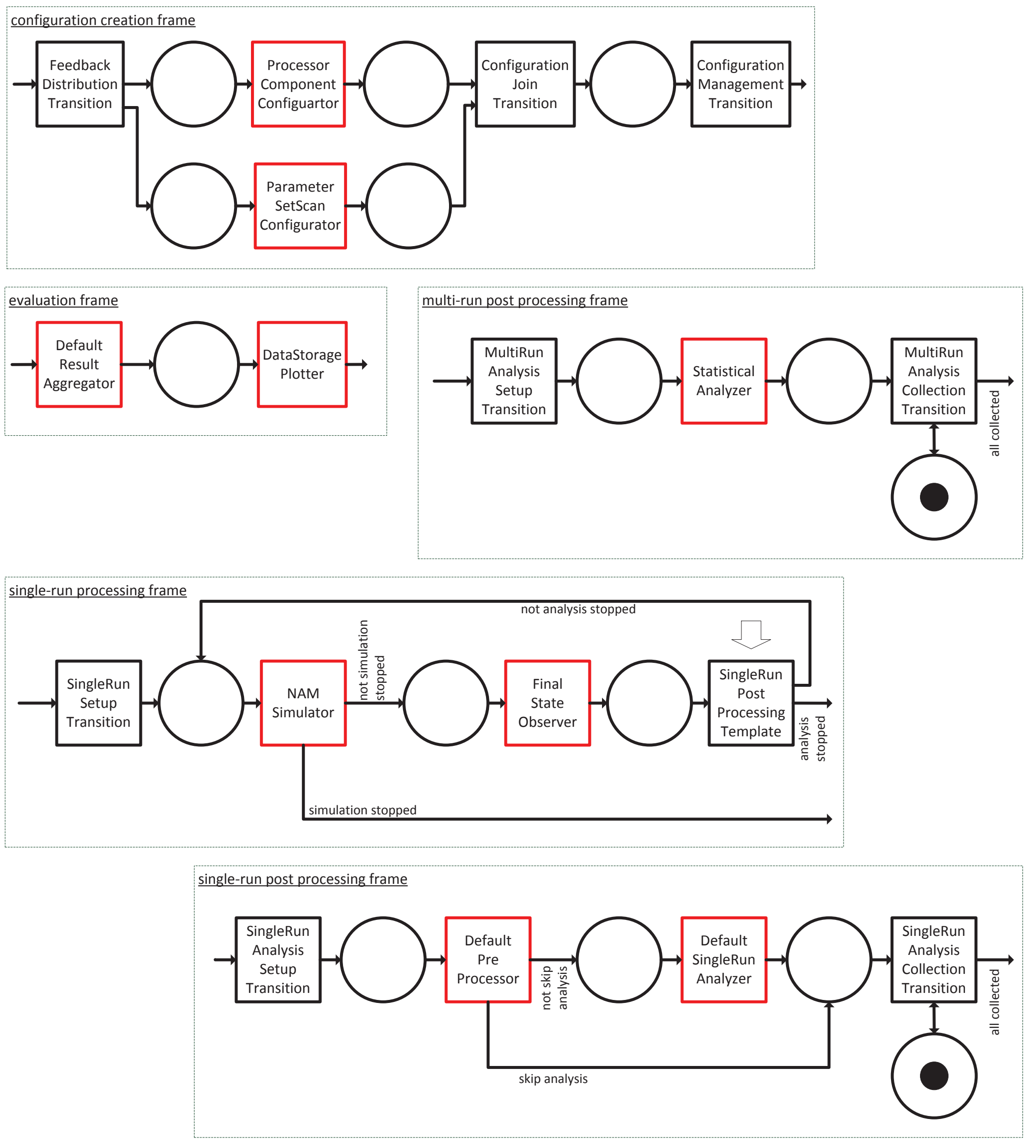

configuration creation frame
Feedback Distribution Transition
Processor Component Configuartor
Configuration Join Transition
Configuration Management Transition
Parameter SetScan Configurator
evaluation frame
Default Result Aggregator
DataStorage Plotter
multi-run post processing frame
MultiRun Analysis Setup Transition
Statistical Analyzer
MultiRun Analysis Collection Transition
all collected
single-run processing frame
not analysis stopped
SingleRun Setup Transition
NAM Simulator
not simulation stopped
Final State Observer
SingleRun Post Processing Template
analysis stopped
simulation stopped
single-run post processing frame
SingleRun Analysis Setup Transition
Default Pre Processor
not skip analysis
Default SingleRun Analyzer
SingleRun Analysis Collection Transition
all collected
skip analysis

Figure 4.10.: Relevant sub-workflows realizing the validation experiment of the NAM method implementation (see Section 3.3.1, pp. 36). Transitions using plugins are represented by red boxes showing the name of the plugin.

4.3.2. Sensitivity Analysis of a Wnt/β-catenin Pathway Model

Figure 4.11 depicts the sub-workflows for executing the sensitivity analysis of the Wnt/β-catenin pathway model (see Section 3.3.2.2, pp. 42). The model is configured by applying the `ConditionalScanConfigurator` that changes each parameter until a given condition is met — in this case a given change in the distance between wet-lab and simulation data (see Table 3.3, pp. 43). Simulator components are again generated by the `ProcessorComponentConfigurator`, as in the validation experiment of the NAM method (Figure 4.10). In this case, however, it produces just one simulator parameter setting, referring to the `OptimizedDirectReactionMethod` that is used for executing the model.

For aggregation and evaluation of parameter setting results, the default method is used again, preparing data for the `SensitivityEvaluator` which calculates the parameter's sensitivities, i.e., changes in the distances between wet-lab and simulation data.

As in the original experiment, multi-run analysis is done by the `TwoSteppedAnalyzer`, and the `TrajectoryObserver` collects data to retrieve the trajectory of βnuc. The `InterpolationPreProcessor` extracts the right time-points from the trajectory to allow a comparison to wet-lab data by the `EuclideanDistanceTrajectoryComparator`.

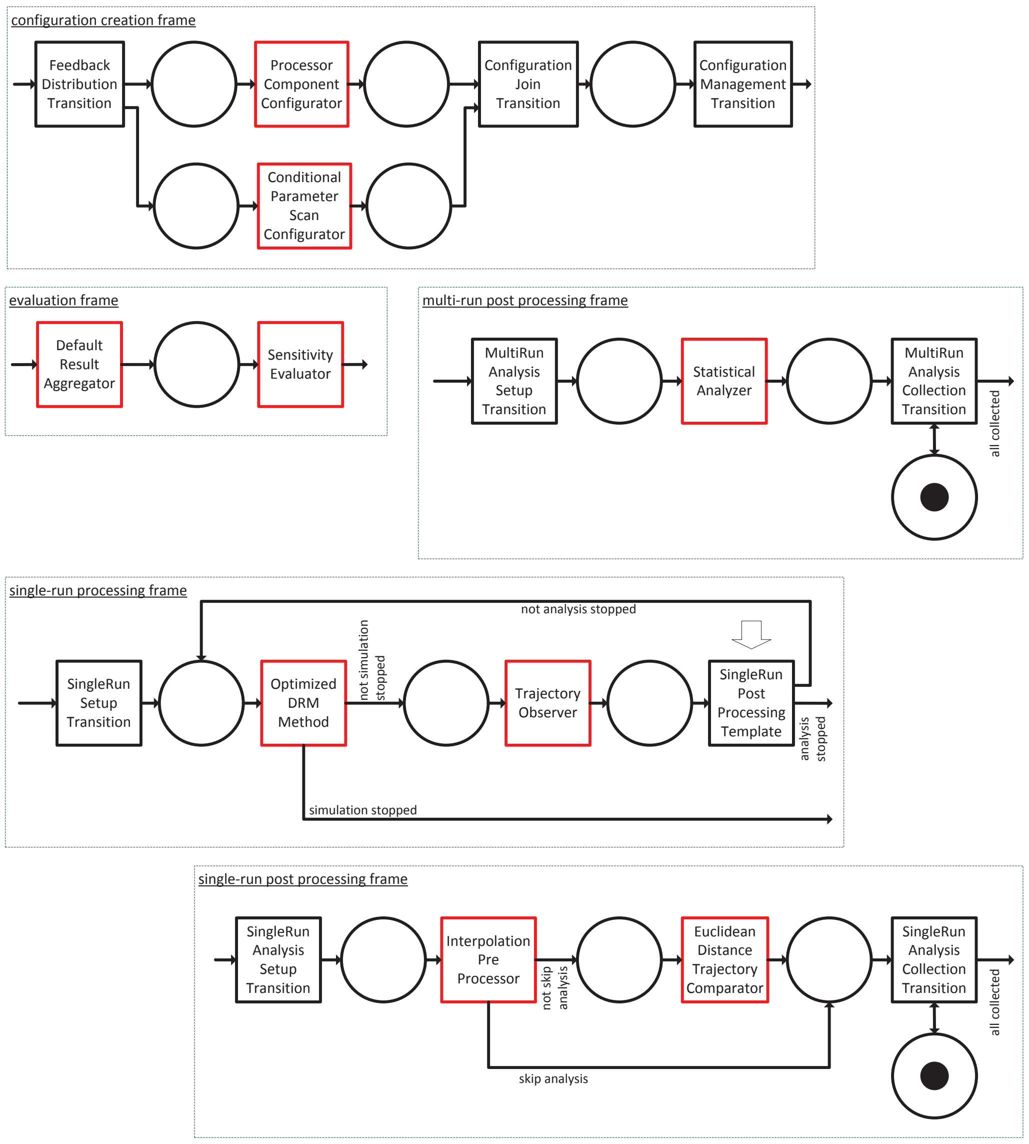

configuration creation frame
Feedback Distribution Transition
Processor Component Configurator
Configuration Join Transition
Configuration Management Transition
Conditional Parameter Scan Configurator
evaluation frame
Default Result Aggregator
Sensitivity Evaluator
multi-run post processing frame
MultiRun Analysis Setup Transition
Statistical Analyzer
MultiRun Analysis Collection Transition
all collected
single-run processing frame
not analysis stopped
SingleRun Setup Transition
Optimized DRM Method
not simulation stopped
Trajectory Observer
SingleRun Post Processing Template
analysis stopped
simulation stopped
single-run post processing frame
SingleRun Analysis Setup Transition
Interpolation Pre Processor
not skip analysis
Euclidean Distance Trajectory Comparator
SingleRun Analysis Collection Transition
all collected
skip analysis

Figure 4.11.: Relevant sub-workflow realizing the sensitivity analysis of the Wnt/β-catenin pathway model (see Section 3.3.2.2, pp. 42). Transitions using plugins are represented by red boxes showing the name of the plugin.

4.3.3. Optimization of a Wnt/β-catenin Pathway Model

The sub-workflows for executing the optimization experiment of the Wnt/β-catenin (see Section 3.3.3, pp. 45) pathway model are depicted in Figure 4.12. Compared to the sensitivity analysis experiment, only the configuration creation and evaluation sub-workflows are changed. The corresponding plugins are reused from the original optimization experiment, i.e., model configuration is done by the `HookeJeevesModelConfigurator`, and evaluation is done by the `ParameterSearchEvaluator`.

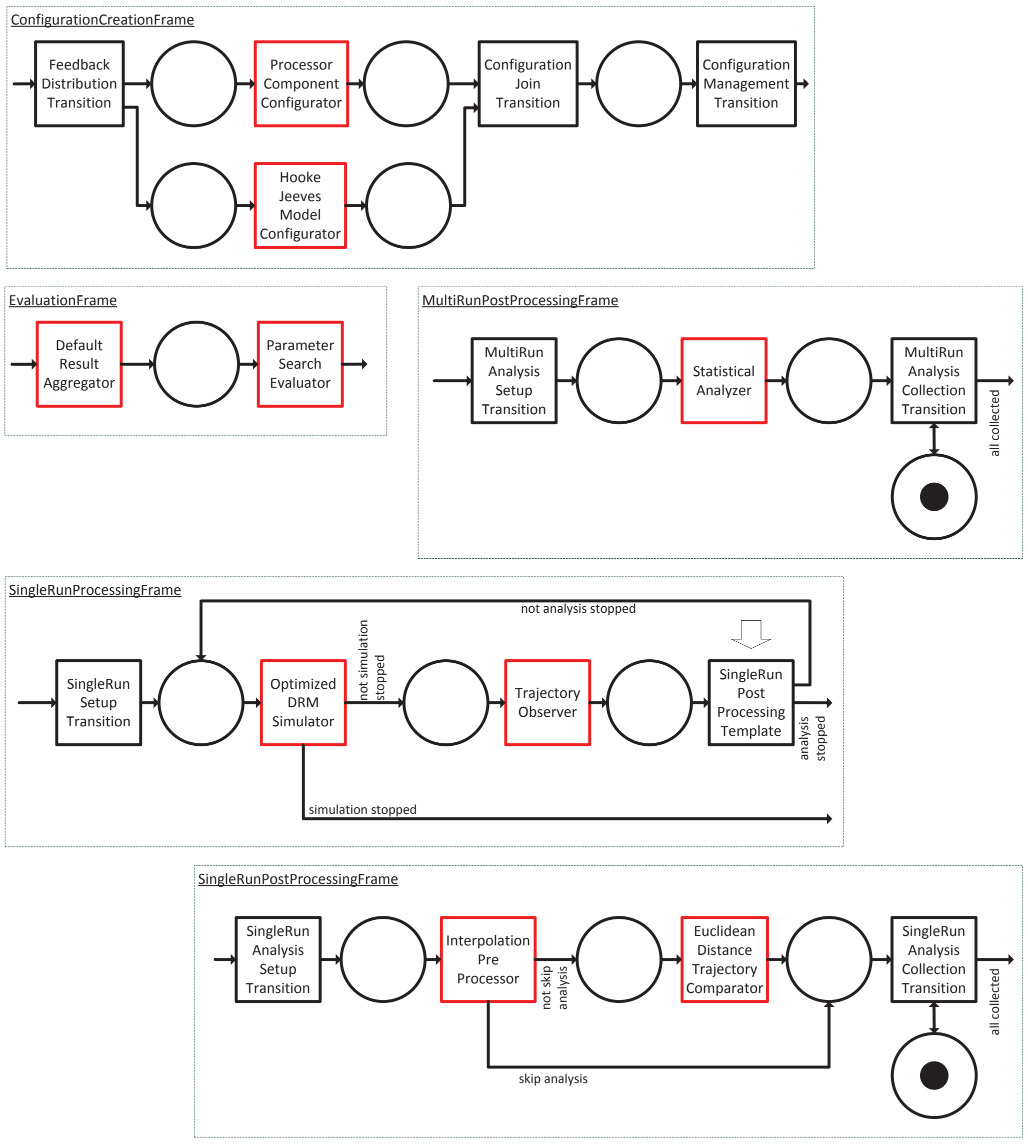

ConfigurationCreationFrame
Feedback Distribution Transition
Processor Component Configurator
Configuration Join Transition
Configuration Management Transition
Hooke Jeeves Model Configurator
EvaluationFrame
Default Result Aggregator
Parameter Search Evaluator
MultiRunPostProcessingFrame
MultiRun Analysis Setup Transition
Statistical Analyzer
MultiRun Analysis Collection Transition
all collected
SingleRunProcessingFrame
not analysis stopped
SingleRun Setup Transition
Optimized DRM Simulator
not simulation stopped
Trajectory Observer
SingleRun Post Processing Template
analysis stopped
simulation stopped
SingleRunPostProcessingFrame
SingleRun Analysis Setup Transition
Interpolation Pre Processor
not skip analysis
Euclidean Distance Trajectory Comparator
SingleRun Analysis Collection Transition
all collected
skip analysis

Figure 4.12.: Relevant sub-workflow realizing the optimization of the Wnt/β-catenin pathway model (see Section 3.3.3, pp. 45). Transitions using plugins are represented by red boxes showing the name of the plugin.

4.3.4. Exploratory Analysis of a VulcaNoCs Model

Figure 4.13 shows the sub-workflows for the exploratory analysis of the VulcaNoCs Model, described in Section 3.3.4, pp. 46. As in all previously described workflows, `ProcessorComponentConfigurator` creates simulator parameter settings. Again, only one setting is produced by this method, referring to the `VulcaNoCsWrapper`. As in the original experiment, this simulation engine executes the whole simulation run in one step, and the `VulcaNoCsResultObserver` collects results that are passed through by default methods for single-run analysis. The simulation stop condition is immediately fulfilled — terminating the simulation run. This means, that the loop in the single-run level realizing simulation steps is only executed once, as the whole simulation run is done in that step. The `ParameterSetScanModelConfigurator` is used for configuring the model.

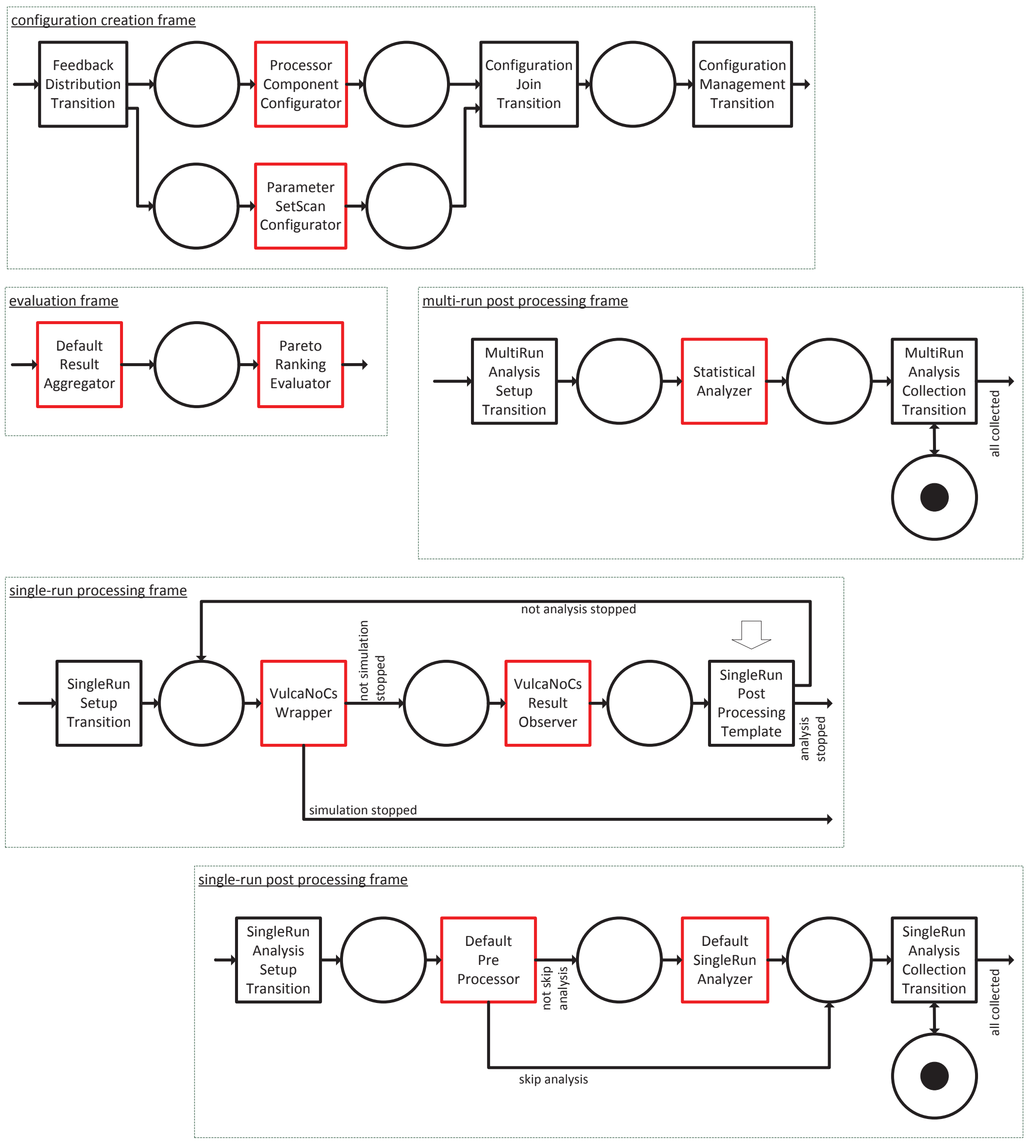

configuration creation frame
Feedback Distribution Transition
Processor Component Configurator
Configuration Join Transition
Configuration Management Transition
Parameter SetScan Configurator
evaluation frame
Default Result Aggregator
Pareto Ranking Evaluator
multi-run post processing frame
MultiRun Analysis Setup Transition
Statistical Analyzer
MultiRun Analysis Collection Transition
all collected
single-run processing frame
not analysis stopped
SingleRun Setup Transition
VulcaNoCs Wrapper
not simulation stopped
VulcaNoCs Result Observer
SingleRun Post Processing Template
analysis stopped
simulation stopped
single-run post processing frame
SingleRun Analysis Setup Transition
Default Pre Processor
not skip analysis
Default SingleRun Analyzer
SingleRun Analysis Collection Transition
all collected
skip analysis

Figure 4.13.: Relevant sub-workflow realizing the optimization Experiment of the VulcaNoCs Model (see Section 3.3.4, pp. 46). Transitions using plugins are represented by red boxes showing the name of the plugin.

The multi-run analysis is realized by the `TwoSteppedAnalyzer`, while parameter setting results are aggregated by the `DefaultResultAggregator` and evaluated by the `ParetoRankingEvaluator`.

4.4. Summary

This chapter showed how the experiment structure, developed in Chapters 2 and 3, has been mapped to workflows.

To realize the experimentation layer as workflow, the workflow management system WorMS, developed by Rybacki et. al. [159], has been exploited. It offers means for describing experiment workflows, experiment documentation, and data provenance. Experiment workflows can be described by the given intermediate workflow representation, i.e., workflow nets, which allows a mapping of other workflow description languages, like XPDL or BPEL. Experiment documentation and data provenance are supported by WorMS' *Monitoring* and *Data Store* components, whose interaction allows tracing and storing of experiment steps realized in a workflow.

A hierarchical workflow description of the experiment is possible with templates and frames. The plugins introduced and described in Section 3.1, pp. 29 could be reused, and further plugins were integrated for data processing, e.g., as used by `ResultAggregationTransition` or `SingleRunAnalysisPreProcTransition`. In most cases default plugins have been employed for data processing, like interpolation for `SingleRunAnalysisPreProcTransition`. However, custom algorithms can be integrated, as well, to give the experimenter additional degrees of freedom.

To show the applicability of the experiment workflow, the example experiments described in Section 3.3, pp. 36 have been mapped to it. The resulting workflow descriptions are straight forward realizations of these experiments, i.e., plugins used in the original experiments are reused and default plugins are used for the additional (data processing) transitions.

Altogether, the workflow realization of the experiment structure executes simulation experiments in a fine-grained manner, where each simulation step is realized as an own transition. This granularity might hamper efficiency, as the output of each activity is stored by *Monitoring*, e.g., the output of a simulation step contains processor and model state, leading to large amounts of data to be stored for complex models. To circumvent this, an alternative single-run (sub-)workflow might be beneficial which relies on one activity executing the whole simulation run. This, however, restricts monitoring, as not every step is tracked anymore. Hence, the user should decide which variant to use, the fine-grained one for better monitoring, or the coarse-grained one, for better performance.

Part II.

Algorithm Selection and Composition

5. Integrating Algorithm Selection and Composition Mechanisms into the Experiment Structure

"Quick decisions are unsafe decisions."

Sophocles

Selecting among a set of algorithms, the one that produces the best result for the problem at hand, can be challenging for users. As simulation experiments comprise different tasks (see Section 2.3.1, pp. 25) such decisions have to be made multiple times during the execution of an experiment. Consequently, decisions in a simulation experiment could be reduced by decreasing the amount of required *manual* algorithm selections for the given (sub-)problems.

Adaptive software [141], i.e., software that adapts to user requirements and to changes in the environment where it is executed, is concerned with applying the right algorithm configuration in the right situation. From the taxonomy of composition in adaptive software, McKinley et al. [133] proposed, a dynamic composition of algorithms should be pursued to achieve user assistance in applying the right algorithms for given problems. This conforms to the perspective of autonomous computing [98] which deals with the (re-)creation of composed algorithms that are better tuned to the tasks at hand. A simulation system complying with such ideas gets aware of its context, i.e., the problems it shall be applied to, via learning and can then optimize itself.

Automatic composition has also been addressed in other fields, but mostly for enabling the execution of new tasks that require interaction between the components (e.g., web services). Such a composition can be supported via languages or language features, like language-integrated queries (LINQ, e.g., see [17]), or by additional tools, like K-BACEE [167] which allows to automatically evaluate component 'ensembles'. These methods, however, are fundamentally different from the approach followed in this work, aiming at the composition of complete algorithms, each *already* designed to fulfill the task at hand.

General purpose tools have been developed using features for selection & composition of algorithms. When formulating the *Algorithm Selection Problem (ASP)*, Rice [155] discussed possible selection mechanisms, which he applied to numerical integration algorithms and operating system schedulers. This led to the development of so-called *problem solving environments*.

Such an environment is Pythia [93], a complex software system for recommending suitable scientific algorithms. The recommendations rely on available machines, algorithms, and past performance results which are stored in a database. For extracting knowledge out of the available performance data, Pythia realizes a *knowledge discovery in databases* methodology. Thereby, performance measurements and experiments can be defined to conduct data mining, and generate selection mappings. While this concept can be seen as pattern for algorithm selection & composition in this work, it does not perfectly fit, as Pythia offers algorithm selection for arbitrary algorithms. Pythia restricts the selection & composition mechanisms to algorithm selection, while the application domain is too general. A generalization of the mechanism and a narrowing of the application domain (to simulation experiments) could lead to better suited results. Furthermore, a seamless integration into a simulation system is required.

In the simulation domain, however, adaptation is typically not realized as a property of the simulation system, but as a feature of specific algorithms. For example, various approaches for adaptive algorithms have been invented in the field of parallel and distributed discrete-event simulation (e.g., [54, 37, 134, 21, 138, 185]). Helms et. al. [79] generalized an adaptation concept for arbitrary simulation engines.

The *Simulation Algorithm Selection Framework (SASF)* [48], developed by Ewald [48], integrates adaptation mechanisms for simulation engines into the simulation system JAMES II. While SASF represents a sophisticated approach for introducing algorithm decision making to simulation experiments, it is restricted in two ways. First, it is not applicable to any other algorithms than simulators, and, second, such algorithms can be composed by algorithm selection only.

Consequently, SASF will be extended to integrate selection & composition mechanisms on arbitrary algorithms that are used in simulation experiments. This chapter will illustrate this extension, starting with a summary of SASF in the first section. Different types of algorithms that are applied during an experiment shall be considered for selection & composition. Therefore, a general problem solver interface for such algorithms is introduced in the second section. Finally, the extension of SASF is presented. It considers arbitrary algorithms implementing the problem solver interface, and goes beyond mere algorithm selection by introducing a synthetic problem solver that allows the integration of different selection & composition mechanisms (e.g., ensemble methods [157], or online adaptation [62]).

5.1. Overview over the Simulation Algorithm Selection Framework SASF

SASF is built on JAMES II, i.e., it uses JAMES II's plugin system and experimentation capabilities, while it can be adapted to work with other host simulation systems, as well. SASF's original design applies the *Algorithm Selection Problem (ASP)* [155] to simulation algorithms, i.e., it considers model features and performance measures of simulation algorithms (retrieved in previous evaluation experiments) to generate selection mappings that map features to algorithms for proposing the most suitable simulation algorithm to execute a model. A more detailed overview over SASF, including a definition of the ASP, will be given in the following.

5.1.1. The Algorithm Selection Problem

The ASP (see Figure 5.1), formally defined by Rice [155], is concerned with finding the most suitable algorithm from a set of algorithms $A \subseteq \mathbb{A}$ in order to solve a problem $x \in \mathbb{P}$. To achieve that, certain features $f \in \mathbb{F}$ are extracted from x, using a *feature extraction mapping* (or *feature extractor*) $F : \mathbb{P} \to \mathbb{F}$. As the problem x is usually a model, features could be, e.g., the number of species in a chemical reaction network, or the population size in a demographic model. Note that in the following extracted features from a problem $x \in \mathbb{P}$ will be written as $f_x \in \mathbb{F}$. Features can be encoded as real numbers, hence, $\mathbb{F} = \mathbb{R}^m$ for m features.

Furthermore, for rating the quality of an algorithm $a \in \mathbb{A}$, a *performance mapping* P is required:

$$P : \mathbb{A} \times \mathbb{P} \to \mathbb{R}^n \tag{5.1}$$

It maps an algorithm $a \in \mathbb{A}$ and a problem $x \in \mathbb{P}$ to an element of *performance measure space* R^n. As multiple performance metrics are possible for an algorithm, the performance measure space has n dimensions. Metrics of an algorithm's performance can be e.g., its speed, its memory consumption, its accuracy, or else. Thus, the quality of an algorithm highly depends on the context in which the algorithm shall be used, and the user has to define the *criteria* on which an algorithm shall be evaluated. Those user criteria can be considered 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:

$$\|p(a; x)\| = g(p(a, x), w) \in \mathbb{R}_{\geq 0} \tag{5.2}$$

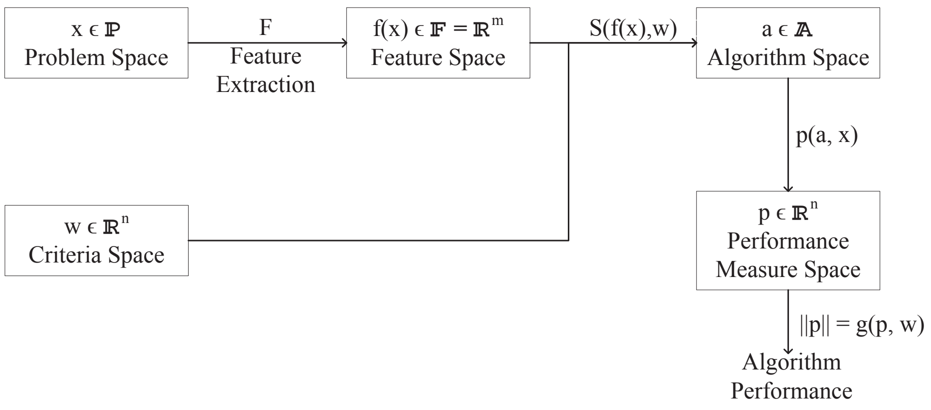


Figure 5.1.: Illustration of the algorithm selection problem (according to [48]) as defined by Rice [155, p. 75].

If, for instance, the user focuses on two performance criteria, like speed and memory consumption, the performance measure space has 2 dimensions. In this case, a criterion vector $w = (1, 2)^T \in \mathbb{R}^2$ would denote that speed is twice as important as memory consumption.

The final and central element of the ASP is the *selection mapping*, which uses the features f_x of a problem $x \in \mathbb{P}$ and the user criteria $w \in \mathbb{R}^n$ to select an algorithm $a \in A$:

$$S : \mathbb{F} \times \mathbb{R}^n \to A \tag{5.3}$$

This equation does not give detailed information about the concrete realization of the selection mapping, but illuminates relevant aspects of the problem, i.e., required structures and the basic task. The concrete selection mapping has to be determined in additional considerations, e.g., by applying machine learning techniques.

5.1.2. Requirements for SASF

SASF has been designed with respect to five technical requirements [48]:

Separation of Concerns Selection can only happen if orthogonal functionality is handled in separated tasks. Thereby, an exchange of components for one task does not influence the other ones; e.g., selecting a simulation algorithm should not lead to a different logging mechanism. In addition, the algorithm selection itself can be regarded as task and a separated treatment facilitates its handling.

Scalability Scalability is important to constrain the computational overhead that results from large amounts of data. For algorithm selection, this includes mostly performance data imposed on algorithms. In the ideal case, the costs for increasing the amount of data and algorithms rise linearly.

(Automated) Behavioral Introspection A software tool realizes behavioral introspection if it is able to collect and evaluate data about its internal algorithms. This includes recording the performance of an algorithm applied on a given problem, storing the performance data, and retrieving the performance data associated with a certain algorithm applied on a problem. The introspection should be automated as much as possible, and user interaction should be minimized. Furthermore, the performance measuring should be as less intrusive as possible, i.e., it should not influence the performance that is monitored.

Meta-Learning / Performance Evaluation Many selection techniques exist (e.g., machine learning, or adaptation algorithms like genetic algorithms) — all comprising certain advantages and disadvantages. Selection mappings, resulting from different techniques, need to be evaluated and selected, as well. Hence, a meta-learning mechanism is required to generate, evaluate, and manage selection mappings with different methods.

Behavioral Intercession Behavioral intercession enables software to be adaptive; i.e., the software may adapt its behavior by altering the selection of algorithms and putting meta-learning into effect. The user should have control over the intercession process. In particular, he should be able to switch it off, e.g., if he knows the best algorithm, is interested in the behavior of one particular algorithm, or knows that the selection mapping is inefficient for the given problem.

5.1.3. The Design of SASF

The design of SASF orients on the described requirements and comprises six basic components that are divided into three layers. Figure 5.2 shows these components interfacing with the host simulation system JAMES II.

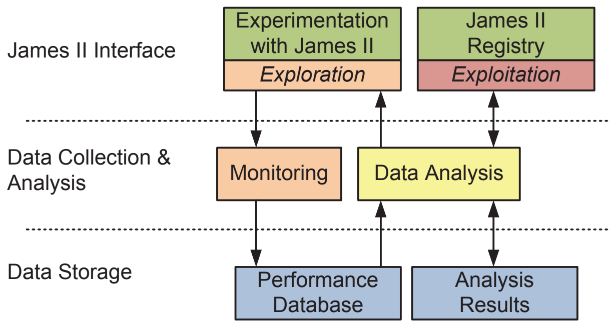


Figure 5.2.: The design of SASF as described by Ewald [48]. Components are sorted according to three layers: interaction with the host system (components exploration and exploitation), data collection (components monitoring and data analysis), and data storage (components performance database and result analysis.

The topmost layer includes the components to explore the performance of JAMES II's simulation algorithms and to exploit results of the following analysis. Thereby, the exploration component provides the data, e.g., by executing performance experiments, while the exploitation component applies solutions, i.e., selection mappings. The middle layer is concerned with monitoring and analyzing data. Monitoring measures the performance of simulation algorithms, which is analyzed to find the most appropriate selection mapping. Both, performance data and analysis results are stored in corresponding components of the bottom layer.

The remainder of this section will give further insight into essential functionality of SASF, by focusing on three tasks that are faced. This includes the monitoring and storing process, the data analysis pursuing suitable ASP solutions by executing performance experiments, and the exploitation in form of an algorithm selection registry developed for JAMES II to apply ASP solutions in JAMES II's plugin selection process.

5.1.4. Monitoring and Storing

The data, which are used to find selection mappings, are monitored and stored during performance experiments. For storing, SASF uses a performance database whose entity relationship diagram is

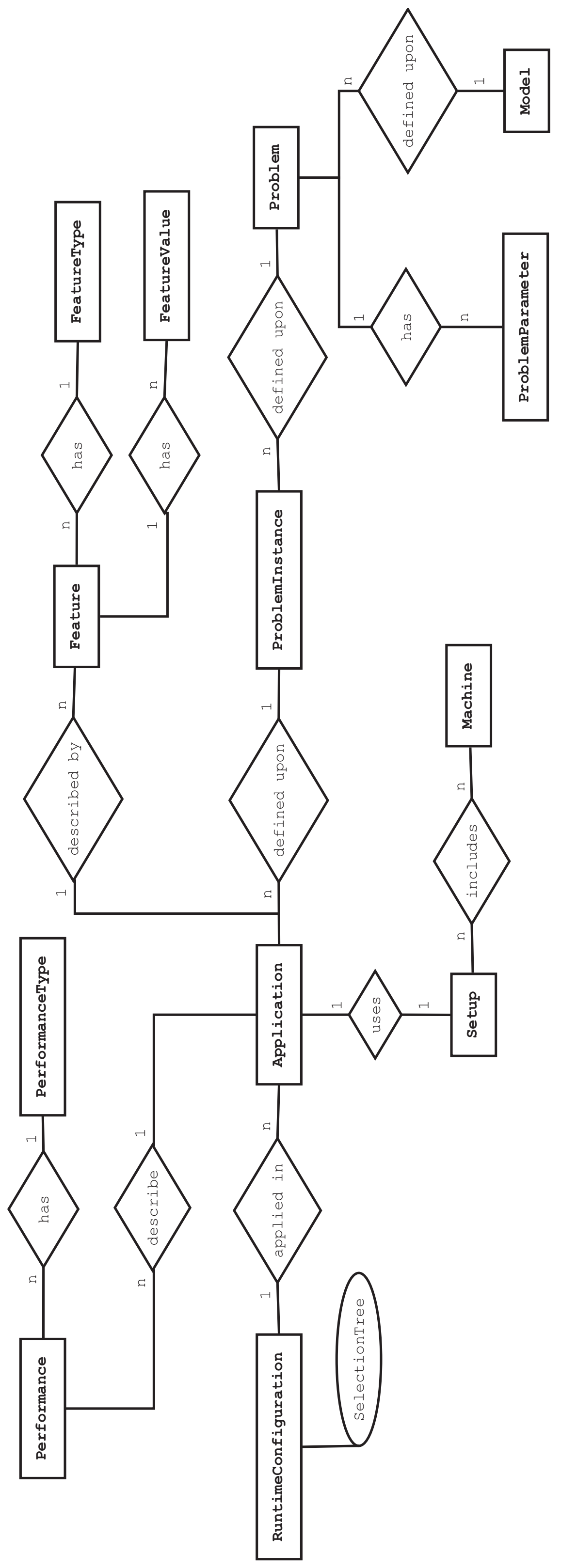


Figure 5.3.: Entity-relationship diagram of the performance database [48]

depicted in Figure 5.3. The mapping of the ASP to the database's components is depicted in Figure 5.4. Selection mappings and performance criteria are not reflected in the performance database, as mappings are generated after performance data have been measured, and criteria are given by the user. The other entities have to be retrieved during the performance experiment and are, consequently, stored in the performance database.

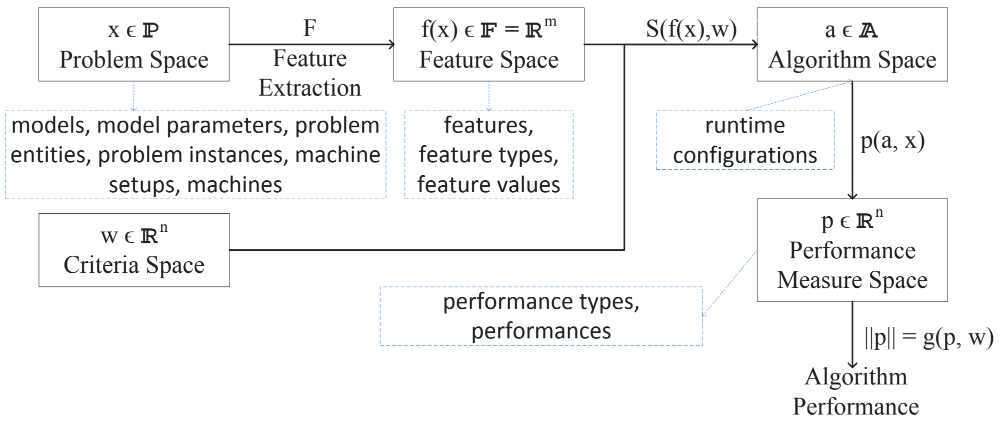


Figure 5.4.: Mapping of ASP (black lined boxes) to the database (blue dashed boxes) components as described by [48].

Problem Space $\mathbb{P}$ As SASF aims at selecting simulation algorithms, the problem space consists of the models that can be simulated. Such models are entities of arbitrary complexity and, hence, storing them directly into a database is challenging. To circumvent this issue, SASF identifies models by URIs (as does the workflow described in Section 4.2) that hold the concrete locations of the models and ensure that a model is accessible, as long as its URI can be resolved.

However, the behavior of a model is not only defined by the model itself. As shown in previous chapters, models usually have parameters that can be altered. Hence, the set of investigated parameter settings, which have been executed during a performance experiment, is stored in the performance database, as well. The combination of model and parameter setting is considered to be a `Problem` entity.

In addition to parameters, many models are influenced by stochasticity — their behavior can vary in different executions, even with the same parametrization. To stochastic factors, an intermediate entity named `ProblemInstance`, containing additional information like the used random seed, is used in the performance database. Hence, for each problem entity a set of problem instances exists.

Finally, the computational setup (i.e., the used machine, operating system, etc.) can be considered to be a dimension of the problem space, as it usually influences the performance of simulation algorithms, e.g., for the runtime of a distributed algorithm, it is important whether it is executed on a distributed setup like a cluster or a single-core Central Processing Unit (CPU). Consequently, `Machine` and `Setup` are entities of the performance database.

Algorithm Space $\mathbb{A}$ To store algorithms, SASF uses the notion of *selection trees*. Those trees reflect the compositional design of simulation algorithms, which, usually, consist of components that are arranged hierarchically, e.g., the NAM simulator of Section 3.3.1 uses a collision detection which can use a stepsize control. A selection tree comprises the factories for all components of the represented simulation engine. For instance, the factory of a stochastic simulation algorithm is the root node of the corresponding selection tree, where the root's child nodes hold the factories for its event-queue and RNG.

In addition to the selection tree, simulation algorithms may have a version to track the development state. Version and selection tree are stored in a `RuntimeConfiguration`.

Feature Space $\mathbb{F}$ For extracting features from a model instance, SASF offers feature extractors implementing the `IFeatureExtractor` interface. It contains the method `extractFeatures` which receives a parameter block containing the problem specification and returns a map of (unique) feature names and corresponding values for the given problem.

Usually, the amount of required features is unknown beforehand, and the application of feature extractors is often restricted to specific model formalisms, e.g., a specific feature for a chemical reaction network is the count of species. Hence, features and feature types might be added during later experiments and a generic storing of them is required.

Feature extractors are integrated into SASF by an own plugin type. Features are stored in the performance database, by defining entities for features identifiers, feature types, and feature values. The factories for feature extractors can be filtered according to their application. To avoid conflicts and redundant information, the performance database ensures that no two features of the same type are applied on the same problem instance (which happens, e.g., when applying two different feature extractors that return similar feature types).

Performance Measure Space Different performance measures for simulation algorithms exist, e.g., computation time, space, or accuracy. Performance measurers used in SASF implement the `IPerformanceMeasurer` interface, including the method `measurePerformance` which receives a runtime information object, that contains various information about a simulation run, e.g., duration, used resources, etc, and returns a floating number representing the measured performance value. Similar to feature extractors, performance measurers are integrated by an own plugin type. Their results can be stored in the performance database using the `Performance` and `PerformanceType` entities.

Application All previously described entities center around the `Application` entity (depicted in the entity-relationship diagram of the performance database, see Figure 5.3). It represents the application of a `RuntimeConfiguration` (i.e., algorithm) on a `ProblemInstance` (i.e., instantiated model), where features are extracted from the `ProblemInstance` and the performance of the `RuntimeConfiguration` is measured. In addition the application holds a relation to the used setup, i.e., computational resources, to support the repeatability of experiments. While the application is not directly referenced in the ASP definition, it represents the central entity where all components of the ASP that are represented in the performance database, are combined. For instance, the `Application` entity might hold the reference to the performance data for applying a stochastic simulator on a species-reaction model, including the extracted count of species as model feature and the duration of the run as performance measure.

Scalability and Generality The performance database has been designed with the goal of generality and scalability. Generality with respect to the host simulation system is ensured, as only three entities (selection trees, feature extractors, and performance measurers) have to be adapted for using other host simulation systems than JAMES II.

To realize scalability with respect to recorded data, a *DataBase Management System (DBMS)* is used that can be exchanged to allow the use of appropriate DBMS for different settings, e.g., operating systems, or amounts of performance data to be collected. To make an exchange possible, the performance database is accessed by an interface and does not depend on a specific implementation. In addition, a plugin type is used to facilitate the integration of alternative database solutions. So far, implementations for the Java DataBase Conectivity (JDBC) [4] and Hibernate [112] have been created.

5.1.5. Creation of Selection Mappings

Based on the monitored and stored performance data, a selection mapping S, also called *selector* in the following, can be generated. It maps (problem) features and user criteria to algorithms for selecting the most suitable one (see ASP, Section 5.1.1). SASF uses the *Simulator Performance Data Mining framework (*SPDM*)* to find selectors. Its overall structure is shown in Figure 5.5. Data

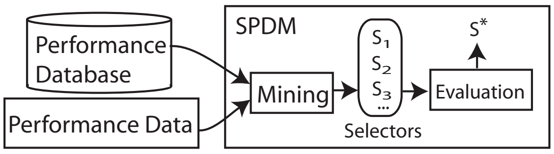


Figure 5.5.: The general scheme of SPDM [48]. Performance data are used for data mining to generate selectors. Those are evaluated and the best one is returned.

mining techniques are applied by selector generators to find selectors that are evaluated in subsequent experiments. Selector generators work on `PerformanceDataSet` objects comprising performance data (extracted from the performance database) and meta-data (e.g., software versions). As many tools exist that can be used for generating selection methods, current selection generators rely on such external tools rather than implementing own mechanisms, e.g., the machine learning tool WEKA [77] is used to realize the decision tree J48 [194], MLJ *(Machine Learning in Java)* [94] is exploited for its ID3 generation of decision trees [151], and the *Java Object Oriented Neural Engine (*JOONE*)* [129] has been used to realize classifiers based on neural networks [20].

The result of the data mining process is a set of selectors implementing the `ISelector` interface. To face the challenges of flexible host systems like JAMES II (it cannot be assured that the selected algorithm is available in the used setting), selectors do not return the most suitable runtime configuration for a set of given features. Instead, they sort configurations at hand according to their suitability for the given problem (features). To realize the sorting, comparison methods implementing

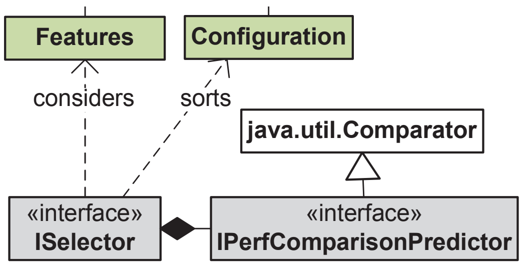
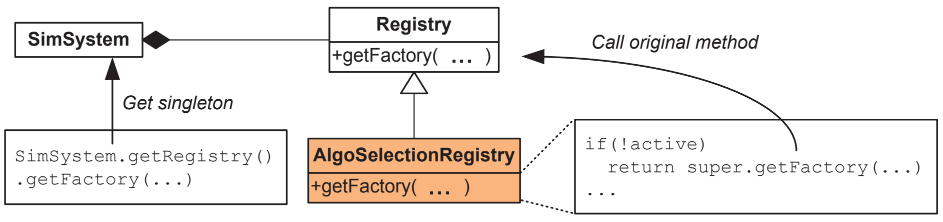


Figure 5.6.: Relationship between `ISelector` and `IPerfComparisonPredictor` interface [48]. The `ISelector` uses an `IPerfComparisonPredictor` to sort configurations according to performance tuples.

the `IPerfComparisonPredictor` interface are used (see Figure 5.6). Such methods compare runtime configurations according to their suitability for the problem at hand.

5.1.6. The Algorithm Selection Registry

To integrate generated selectors, SASF extends the JAMES II registry for automated algorithm selection. It overrides the method `getFactory`, which returns a concrete factory for a given abstract factory and a set of parameters (see Section 3.1.1, pp. 29 for the standard process). In the extended algorithms selection registry, the method picks a selector, retrieves the sorted list of runtime configurations, and returns the first configuration in the list that is available. If this more complex functionality is not required, it can be easily switched off (by setting a flag), leading to the behavior of JAMES II's standard registry (see Figure 5.7).



```
SimSystem.getRegistry()
.getFactory(...)
```

```
if(!active)
  return super.getFactory(...)
...
```

Figure 5.7.: The `AlgoSelectionRegistry` extending the JAMES II registry [48]. It overrides `getFactory` and can still be used as standard registry by setting the `active` flag to false.

Furthermore, the algorithm selection registry comprises functionality for deploying selectors. Therefore, they are associated to the plugin type they apply to, and enriched with meta-data, e.g., the current phase in the plugin's life cycle (e.g., untested, tested, stable, or broken) to allow different failure tolerances (e.g., accepting all, stable, tested, or untested plugins).

Integrating selectors To enable the integration of selectors for a plugin type, users of SASF can add annotations to the corresponding abstract factory. This is necessary as not all plugin types are suitable for the selection process (e.g., model formalisms). Parameters of the annotation can be used for further specification of the selection, e.g., for describing which part of a runtime configuration shall be defined by automatic selection (the whole simulator, or only a component like its event queue).

Multiple selectors can be considered for a single plugin type. Each is handled by a `SelectorManager` that decides whether the corresponding selector is able to select an algorithm for the given problem, extracts problem features, and applies the selector to the problem.

5.2. An Interface for Problem Solvers in Simulation Experiments

To unify selection & composition mechanisms for the different kinds of methods, used in a simulation experiment (e.g., simulators, analysis, or parameter configuration methods), a general interface, for algorithms realizing such methods, is required. These algorithms have to be integrated as plugins and triggered by activities of the experiment workflow (described in Section 4.2, pp. 56) to combine workflow and selection & composition mechanisms in one unifying environment — GUISE. Figure 5.8 illustrates the position of the problem solver interface between workflow and selection & composition mechanism. The *problem solver* — implementing the interface — is the central element realizing methods to be applied during an experiment. The structural part is covered by the experiment workflow, which applies problem solvers for different tasks, whereas the selection & composition learns on performance results of problem solvers. The goal of the learning process is to compose an advanced problem solver for problems with certain characteristics (e.g., time series of a given length), that can be used by the workflow as if it were a standard method.

The problem solver interface has to be sufficiently general for being applicable to the different tasks of a simulation experiment (see Section 2.3.1, pp. 25), whereas it should reflect the specific mechanisms that methods in simulation experiments have in common, e.g., they usually solve problems iteratively.

Typical tasks for which users need assistance in method selection are the configuration of model parameter settings and the analysis of simulation output data. For instance, model parameters can be configured by parameter search algorithms, of which a plethora exists, e.g., [158, 113, 90, 110].

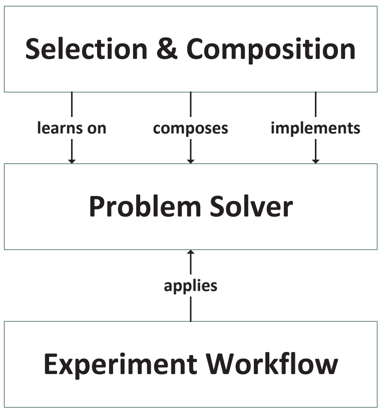


Figure 5.8.: Relation between problem solver, workflow, and selection and composition mechanism. The workflow applies problem solvers to handle tasks. Selection and composition learn on the performance data of problem solver executions to compose them. To make the composition available the mechanism is implemented as a problem solver itself.

Such algorithms produce parameter settings that are evaluated through simulation. The aggregated simulation results are regarded as evaluations of an (unknown) objective function and are returned to the search algorithm as feedback to generate new promising parameter configurations. As another example, steady state estimators — which are often applied during the analysis of simulation output data — produce an estimate for the steady state statistic (e.g., the steady state mean) of a time series generated during a simulation run.

Similar to parameter search, various methods have been proposed for steady state estimation, e.g., in [45, 193, 166, 27, 190, 120]. If an estimator indicates that the time series does not contain sufficient data for the estimation, e.g., due to a warm-up period [10] being not over yet, the simulation run is continued and additional output data are collected until a (sufficiently accurate) estimate can be made. The same applies for various other simulation output analysis methods, like cycle detection methods.

Simulation algorithms, which produce the output data for later analysis, work iteratively, as well. In each iteration (or simulation step), such algorithms calculate the next time step and the associated model state.

These and many other methods applied during simulation experiments, follow an iterated scheme, which only differs in task-specific inputs (e.g., values of an objective function in parameter search, or time series in steady state estimation) and task-specific outputs (e.g., an optimal configuration in parameter search, or an estimated steady state mean in steady state estimation).

5.2.1. Defining the Problem Solver Interface

The interface, for iterative problem solving by algorithms used in simulation experiments, comprises two basic methods: *init* is responsible for initializing the problem solver, i.e., setting it up for the problem solving process; *solve* realizes the problem solving process itself.

Problem solving is usually based on a simple input/output behavior, i.e., the problem is the input and the solution or result is the output. An example for this is a steady state estimator, which estimates a steady state statistic (e.g., mean) of a given time series. The input are the available data points of the time series for which the steady state statistic, which is the output, has to be estimated.

This input/output behavior is mirrored in the *solve* function's signature, which, given a problem space $\mathbb{P}$ (e.g., the set of possible time series) and a set of possible results RES (e.g., the possible estimated steady states), *could be* described by:

$$solve : \mathbb{P} \rightarrow RES \tag{5.4}$$

This signature, however, does not incorporate iterated problem solving that is necessary in cases where not all required information about the problem might be initially available. In such cases, problem solvers need the ability to request further information to handle the problem in additional iterations[1].

To consider problem iterations, instead of static problems, a problem solver may need a state for storing (partial) results of previous iterations, e.g., many steady state estimators keep track of previous results to avoid recalculations. The state of the previous iteration can be considered as input, and the resulting state of the current iteration as output. Consequently, the state includes all information that have to be available for the problem solver: *algorithm state information (AS)* comprising all relevant information for the next iteration, *result information (RES)* representing the solution of the current iteration, and *request information (REQ)* representing additional data required for the next iteration. The set of *problem solver states* $\mathbb{S}$, thus, can be defined as:

$$\mathbb{S} = \{(as, res, req) | as \in AS, res \in RES, req \in REQ\} \tag{5.5}$$

Altogether, given a set of problem solver states $\mathbb{S}$, the *solve* function of an iterated problem solver PS has the following signature:

$$solve : \mathbb{P}_{it} \times \mathbb{S} \rightarrow \mathbb{S} \tag{5.6}$$

Iterated problems from $\mathbb{P}_{it}$ include the initial problem description, and a *request-answer history* $H_R \in \mathbb{H}$. The history is a sequence $H_R = h_1, \ldots, h_n$ with $h_i \in (REQ, ANS)$, i.e., it basically represents a list of issued requests from REQ and corresponding answers from the set of possible answers ANS comprising additional information about the problem (like additional data points of a time series).

The set of iterated problems $\mathbb{P}_{it}$ is, hence, defined by:

$$\mathbb{P}_{it} = \mathbb{P} \times \mathbb{H}, \tag{5.7}$$

i.e., a $p_{it} \in \mathbb{P}_{it}$ comprises the initial problem and its request history holding all request-answer pairs that have been created during previous iterations.

To generate answers for requests, an *answer function* η is applied:

$$\eta : REQ \rightarrow ANS \tag{5.8}$$

The answer function is not part of the problem solver itself, but rather a part of the problem description and consequently depends on the kind of problem to be solved. For instance, the data points of the problem time series of a steady state estimation are usually generated by simulation, making the simulation engine the answer function.

5.2.2. Managing the Problem Solving Process

To solve a problem effectively, it has to be ensured that requests as well as answers are included in the problem definition for the next iteration. To make that happen, a management component (e.g., a predefined workflow) is necessary that interacts with the interface of the problem solver PS and the answer function η.

Figure 5.9 shows a sequence diagram for all three components when solving a problem $p_{it} \in \mathbb{P}_{it}$, iteratively. Figure 5.10 illustrates the process at the example of steady state estimation.

During the initialization of the problem solver PS, an initial state $s_0 \in \mathbb{S}$ is generated. Similarly, the representation of the first problem iteration, $p^0_{it} \in \mathbb{P}_{it}$, only includes the problem itself and an empty request history; it is created by initializing the answer function η.

[1]Problem solvers that do not work iteratively can be treated as iterative problem solvers needing only one iteration.

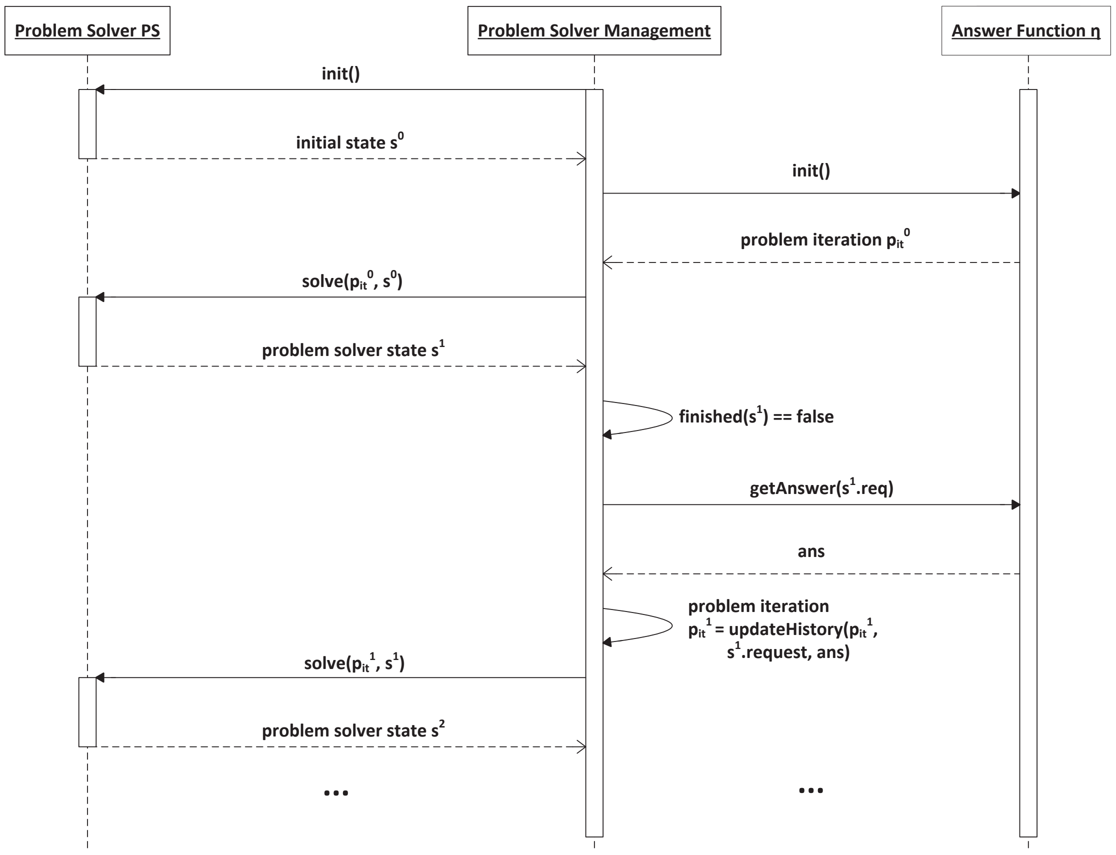


Figure 5.9.: Sequence diagram depicting the iterated communication between answer function (Equation 5.8), e.g., representing simulator, and a problem solver, e.g., steady state estimator. The communication is controlled by a managing component that is also responsible for updating the problem description $p \in \mathbb{P}_{it}$.

With both elements, problem solver state s_0 and problem iteration p_{it}^0, the `solve` method (Equation 5.6) of the problem solver is called. It returns a successor state holding the request for getting additional information about the problem, e.g., in steady state estimation, this would be a boolean value denoting that additional data points of the time series are required. Afterward, the answer function η (Equation 5.8) is called by the management component. The resulting answer (an additional segment of the time series in the steady state estimation example) in combination with the request is added to the request history of the problem, leading to a new problem iteration, which now includes the initial problem description and a request history comprising the first request-answer pair.

After each iteration, the management component checks whether a final problem solver state is reached, by calling the method `finished`. This function highly depends on the used problem solver, respectively the problem at hand. The whole process is repeated until the function returns true. In the steady state example, this is the case as soon as the estimator could compute a the steady state mean estimate.

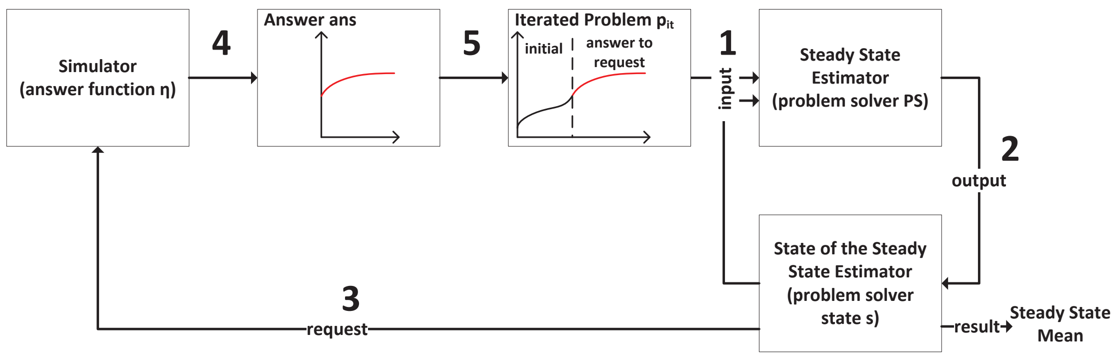

Simulator
(answer function η)
4
Answer ans
5
Iterated Problem p_{it}
initial
answer to request
1
input
Steady State Estimator (problem solver PS)
2
output
State of the Steady State Estimator (problem solver state s)
3
request
result
Steady State Mean

Figure 5.10.: Example scheme for managing the communication between a simulator producing time series for a steady state estimator. The steady state estimator works on its previous state and a time series, generated by the simulator, as input (1). After a solving iteration has been finished, the estimator returns its state (2), comprising a request for the simulator (3), which executes the next simulation steps (4). Thereby, a new segment of the time series is generated and added to the problem time series (5), which is used in the next estimation iteration.

5.2.3. Mapping Methods to the Problem Solver Interface

The problem solver interface can map different methods that are typically used during simulation experiments. This shall be illustrated by an exemplary mapping of four method types. Three of them that have been used during example experiments of Section 3.3. This includes parameter search methods (e.g., Hooke and Jeeves parameter search as used in Section 3.3.3, pp. 45), statistical analysis (e.g., sample mean estimation as used in Section 3.3.1, pp. 36 and 3.3.2, pp 41), and time series comparison (e.g., Euclidean distance comparison as used in Section 3.3.2, pp. 41). In addition, steady state estimation will be mapped to the interface. Table 5.1 gives an overview over all four mappings.

Problem Solver Interface	**Parameter Search Counterpart**	**Statistical Anlz. Counterpart**	**Time Series Comp. Counterpart**	**Steady state Estim. Counterpart**
problem space $\mathbb{P}$	model space $\mathbb{M}$	sample space SA	input time series space $\mathbb{R}^n$	input time series space $\mathbb{R}^n$
result set RES	power set $2^{\mathbb{I}_M}$ of parameter setting space	statistic domain ST	real numbers $\mathbb{R}$	real numbers with $null$ element $\mathbb{R} \cup \{\perp\}$
request set REQ	parameter setting space $\mathbb{I}_M$	natural numbers (with 0) $\mathbb{N}_0$	$\emptyset$	$\{true, false\}$
answer set ANS	objective value set $\mathbb{O}_M$	sample space SA	$\emptyset$	input time series space $\mathbb{R}^n$
algorithm state set AS	state space of the concrete algorithms			

Table 5.1.: Components of the problem solver interface and their counterparts when realizing parameter search, replication estimation, trajectory comparison, and steady state estimation.

Parameter Search Algorithms A parameter search method searches a parameter setting space $\mathbb{I}_M$, by systematically 'probing' it. Probing means that the method requests an objective value $o(p_1, ..., p_n) \in \mathbb{O}_M$ for a parameter setting $(p_1, ..., p_n) \in \mathbb{I}_M$. The strategy of finding interesting parameter settings results from the concrete implementation of the method, e.g., Hooke and Jeeve, particle swarm etc. The objective value is generated by an objective function $f_o : \mathbb{I}_M \rightarrow \mathbb{O}_M$, which is, usually, realized by simulating the problem model. Results of that function are used as hints for generating additional parameter sets and, consequently, finding one or more parameter settings whose objective values meet a set of goal conditions G. These settings are considered to be the result of the algorithm.

This behavior can be directly mapped to the problem solver interface PS. The problem space $\mathbb{P}$ maps to the set of models $\mathbb{M}$ that might be investigated by parameter search. In each iteration, the search method requests objective values for a generated parameter setting $(p_1, ..., p_n) \in \mathbb{I}_M$. Hence, $\mathbb{I}_M$ can be directly mapped to the request set REQ of the problem solver PS. The objective value $o(p_1, ..., p_n) \in \mathbb{O}_M$ is the answer to such requests. Consequently, $\mathbb{O}_M$ equals to the set of answers ANS, and the objective function f_o realizes the answer function η. The set of algorithm states AS results from the concrete implementation and parametrization of the search algorithms. As the result of a parameter search algorithm is a set of parameter settings, the power set $2^{\mathbb{I}_M}$ of the parameter space corresponds to the results set RES.

Statistical Analyzers for Replication Estimation In many cases a statistic, e.g., mean or variance, is calculated during multi-run analysis (like done by the `TwoSteppedAnalyzer`, see Section 3.3, pp. 36). The amount of required (simulation run) replications has to be estimated. A method providing such functionality (called statistical analyzer in the following) takes an initial sample X of data points from the space of possible samples SA as input and estimates a statistic $st \in ST$. It, furthermore, estimates the amount $n_i \in \mathbb{N}_0$ of additionally required data points for getting a reliable estimate of st, e.g., by considering a desired confidence c and allowed error e. If $n_i = 0$ the algorithm terminates with the current estimation of st being the result, otherwise it waits for an additional sample of size n_{i+1} that is used for the next estimation step.

For mapping this kind of method to the problem solver interface, the initial sample is considered to be the initial problem. Hence, SA conforms to $\mathbb{P}$. The amount of additionally required replications is the request, i.e., the set of requests REQ equals the set of natural numbers (with 0), $\mathbb{N}_0$. The answer to a request is the sample of data points created during the replications. Hence, the answer function η returns samples that are usually generated through model execution and single-run analysis. Consequently, the answer set ANS is equal to the sample space SA. Similar to parameter search, the set of algorithm states AS depends on the concrete implementation and parametrization of the used algorithms. The result of the algorithm is the estimated statistic, i.e., its result set RES is the set of possible statistic values ST.

Time Series Comparison A time series comparator gets an input time series $\vec{y} \in \mathbb{R}^t$ which is compared to a reference time series (that can be considered as parameter of the comparator). As the size of the input time series should match the size of the reference time series, the duration t of the model execution that generates the input time series is known from the beginning. Hence, time series comparison does not require an iterated execution and no additional requests are necessary. Consequently, the problem space $\mathbb{P}$ equals the set of possible time series $\mathbb{R}^t$, while the sets of possible requests REQ and answers ANS are empty. As the time series distance is typically a real number expressing similarity, the result set RES of a time series comparator is the set of positive real numbers $\mathbb{R}^+$.

Steady State Estimation A steady state estimator [7, p. 96] takes a time series $\vec{y}^0 \in \mathbb{R}^n$ as input, and returns a steady state statistic — usually the mean $\hat{S} \in \mathbb{R} \cup \{\perp\}$, where $\perp$ denotes a null element, i.e., no estimate for the mean could be given. In this case, the time series is too short to allow a proper estimation (since the warm-up phase is too long), and additional data are required. To request additional data, the estimator returns a boolean value. If it returns $true$, it awaits additional data, i.e., the following time series segment which is also a series of real numbers $\vec{y}^1 \in \mathbb{R}^n$.

Considering a steady state estimator as problem solver, the problem space $\mathbb{P}$ is the set of possible time series $\mathbb{R}^n$. The set of requests REQ equals the set of boolean values $\{true, false\}$, while the set of answers ANS can be mapped to the set of input trajectories $\mathbb{R}^n$. The result set RES of a steady state estimator is the set of real numbers with null element $\mathbb{R} \cup \{\bot\}$. Similar to the previously described methods, the set of algorithm states AS depends on the concrete algorithm.

5.3. Extending SASF for Decision Making on Problem Solvers in Simulation Experiments

SASF applies algorithm selection techniques on simulation algorithms, but, so far, it does not support arbitrary algorithms following the problem solver interface introduced in Section 5.2.

This section describes the generalization of SASF for such algorithms, and for integrating different selection & composition strategies, in addition to algorithm selection. Therefore, further requirements are identified, that have not been considered in original SASF (see Section 5.1.2). Afterward, an extended structure of SASF is presented, followed by detailed descriptions of additional or adapted components.

5.3.1. Requirements

SASF fulfills the requirements for a simulation algorithm selection tool, listed by Ewald [48], and subsumed in Section 5.1.2. In addition, a tool inducing selection & composition mechanisms on arbitrary experiment methods has to be *flexible*, as it should be able to handle different algorithm types corresponding to the problem solver interface proposed in Section 5.2.1.

As described, the problem solver interface represents iterative algorithms that request for additional information about the problem during runtime. Since these information are gained constantly during the process of solving the problem, it is not sufficient to extract features only once from the initial problem description $x \in \mathbb{P}$ but from the iterated problem $x_{it} \in \mathbb{P}_{it}$. Hence, the standard problem feature extractors, which are applied on the initial problem description, might not provide all relevant data. For instance, in steady state estimation, new segments of the problem time series are investigated during each iteration, resulting in a change of the extracted feature values during runtime, e.g., the running mean might change with more data points.

This runtime feature extraction is trivial for problems where the initial problem representation and the information gained during the problem solving iterations have similar data types, and the feature extractors of the initial problem representation can be reused. For instance, the initial problem of a steady state estimation is a time series, and the additional information are time series segments—both represented by lists of numbers. However, cases exist where the type of the initial problem and the type of the data gained in later iterations differ. An example for this is parameter search, which, usually, works on a model as initial problem and gains additionally information in terms of objective values. Consequently, feature extractors need to be *flexible* enough to work on different problem types, and on additional information that are generated during the problem solving process.

An additional problem is the change of problem solvers, respectively their states, during the course of the solving process. Problem solvers are not static, i.e., they do not create one result for one given (initial) problem representation, but, rather, produce different results as they go through the problem solving iterations and gain more information about the problem at hand. Consequently, it is not sufficient to measure the performance of a problem solver just once, but on different stages of the solving process, e.g., after a steady state estimator worked on different segments of a time series. The resulting performance can then be considered as feature by the selection & composition mechanism. Thus, it is beneficial to have feature extractors for problem solver states.

As it is not sufficient to consider initial problem features and decide what problem solver performs best, like in the standard ASP, the selection & composition process has to be flexible, as well. Different strategies might be applicable, like ensemble learning [157] or online adaptation [62], to incorporate features of problems and problem solvers that are extracted during runtime. Hence, instead of only applying algorithm selection, the SASF extension should allow the integration of such strategies for composing algorithms and combining their results to better versions (of course, it might exploit algorithm selection techniques, as well).

In addition to being flexible, the selection & composition mechanism has to be easily *accessible* to users, as it shall help users in applying problem solvers. In the ideal case, the result of this mechanism should be as easily applied as any problem solver it works on. For instance, a selector created with the standard SASF framework should be treated as a simulation engine from the users viewpoint. The user should be able to apply the selector on the model he wants to simulate, while the selector automatically selects the 'right' simulation engine and uses it to perform the simulation run. The user should not have to care about how the selection mechanism works and how it is integrated into the tool. For general selection & composition mechanisms, a structure has to be created, that conforms to these different mechanisms and makes their results seamlessly available.

5.3.2. Basic Scheme of the SASF Extension

As described in Section 5.1.3, SASF includes elements for exploring, monitoring, analyzing, and storing the performance of algorithms, as well as storing and exploiting analysis results (see Figure 5.2, pp. 72).

In order to extend SASF for handling different algorithm types, these elements have to be adapted. Instead of using selectors, a *Synthetic Problem Solver (SPS)* implements the problem solver interface and incorporates the selection & composition process. The SASF registry is enriched with a mechanism for deploying *SPS* instances.

For a systematic exploration of the problem space, a *problem generator* is introduced, that generates training data for the selection & composition process. The performance database and monitoring mechanisms are reused with some adaptations to ensure flexibility.

The interplay between the elements is depicted in Figure 5.11. For exploration, a set of *base-line*

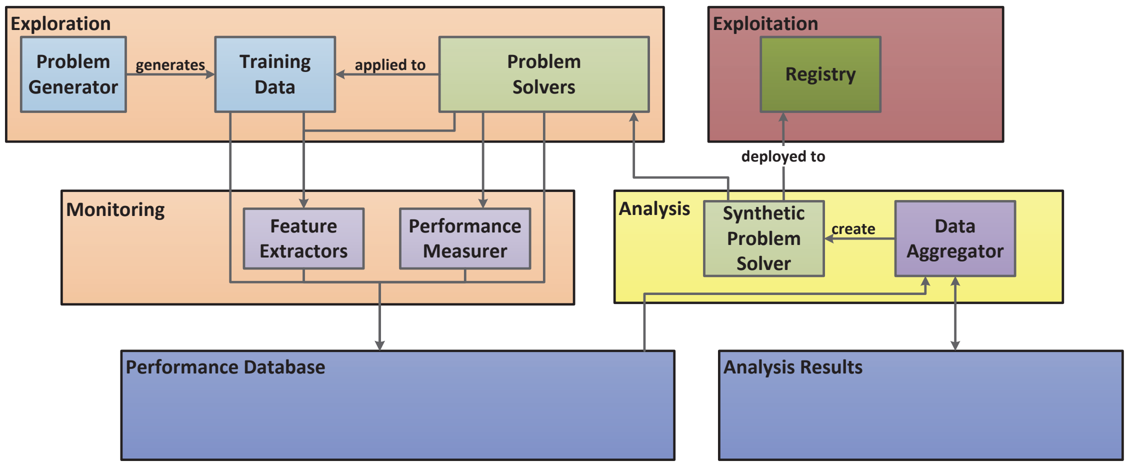


Figure 5.11.: The SASF extension — the basic scheme of SASF remains untouched, while the inner workings are adapted.

problem solvers are applied to training data generated by the the problem generator. The monitoring of the application process remains unchanged, i.e., feature extractors, and performance measurers retrieve data, that are stored in the adapted performance database. Based on such data, a selection & composition mechanism is trained, e.g., a decision tree. This mechanism is the central element of an *SPS* instance, which furthermore, comprises the used problem solvers and feature extractors. Each *SPS* instance is considered to be of the same type as the problem solvers it is based on, e.g., an *SPS* instance based on steady state estimators is considered a steady state estimator itself. Hence, the problem solvers in the exploration can be synthetic problem solvers, as well, which allows meta-learning i.e., learning on different synthetic problem solvers incorporating different selection & composition mechanisms. To exploit SPS instances, they are deployed to an adapted algorithm selection registry, which automatically generates according factories. Hence, such instances are accessible to the user through the standard plugin system he is familiar with, when using JAMES II.

In the following, a deeper insight into synthetic problem solvers is given, by presenting a formal definition, an algorithmic description, and an example. In addition, the SPS definition is mapped to existing selection & composition strategies, to show its flexibility. Afterward, the SPS implementation is described, and the tools for generating and deploying it are characterized.

5.3.3. Synthetic Problem Solvers

In this subsection, a general structure for the automated composition of problem solvers in a new synthetic problem solver is proposed. The composition happens to achieve superior performance on certain problem classes. The SPS is designed to support different composition strategies for problem solvers and making them accessible to users. Hence, the user now applies an SPS following the problem solver interface, instead of selectors (which even though might be used by the SPS as selection & composition mechanism). The SPS holds and manages the main elements to orchestrate a set of given problem solvers. As it is designed to be an improved version of its base-line problem solvers, it implements the problem solver interface (see Section 5.2), as well.

This subsection illustrates the SPS by presenting its formal description and its technical realization. An example SPS shows the practical application and a mapping to existing composition approaches indicates its generality.

5.3.3.1. Formal Definition

A synthetic problem solver SPS has the following structure:

$$SPS = \langle A, \mathcal{F}_P, \mathcal{F}_S, \varsigma, \kappa \rangle \tag{5.9}$$

i.e., it is defined by a set of problem solvers (A), sets of problem feature and state feature extractor functions ($\mathcal{F}_P$ and $\mathcal{F}_S$), a selection function (ς), and a composition function (κ). In the following, the different components and their interplay are defined on the notation introduced by Rice [155].

Let A be the set of available problem solvers (algorithms) that implement the described problem solver interface. To realize a decision making procedure on problem solvers, a retrieval of relevant features of those solvers and the problem at hand is required. For example, during a steady state estimation with multiple baseline estimators, problem features might characterize the time series, whereas solver features might represent the results of the estimators when applied to that time series. Therefore, departing from Rice's original definitions, two kinds of feature extraction functions are defined. A *problem feature extractor* $F_P : \mathbb{P}_{it} \to \mathbb{F}_P$ extracts a solver-independent feature $f_p \in \mathbb{F}_P$ from the iterated problem, whereas a *state feature extractor* $F_S : \mathbb{S} \to \mathbb{F}_S$ extracts a solver-*dependent* feature $f_s \in \mathbb{F}_S$ from the state $s \in \mathbb{S}$ of a solver $a \in A$. For instance, in the case of steady state estimation, a problem feature extractor extracts variance and range from a request history that represents the generated time series segments. A state feature extractor could return a boolean that indicates whether an estimator detected the end of the warm-up phase. $\mathcal{F}_P$ denotes the sets of available problem feature extractors, $\mathcal{F}_S$ denotes the set of available state feature extractors.

In some cases, it is not sufficient to just consider the features of the current iteration. For instance, if a problem solver always produced bad results during previous iterations, this could be relevant for decision making. Therefore, a *feature history* H_F is introduced, containing all problem and state features that have been previously extracted[2]. It can be formally described by:

$$H_F = (h_1^f, \ldots, h_r^f), \tag{5.10}$$

[2]Note that the feature history is different to the request history of the problem solver interface definition (Equation 5.7) as it is part of the problem solver state rather than its input.

where $h_i^f \in \mathbb{F}_P^n \times \mathbb{F}_S^m$, with $n = |\mathcal{F}_P|$ and $m = |\mathcal{F}_S| \cdot |A|$. Hence, the feature history is essentially a list of r tuples, where r is the number of executed iterations and each tuple contains the problem and state features that have been extracted during the corresponding iteration. This allows a decision-making component to consider the entire history of the ongoing problem solving procedure.

After the features have been extracted, the decision making process can be executed. Two functions are responsible for this, realizing base-line solver selection and composition. Both are defined upon the set of all possible feature histories, $\mathbb{H}_F$:

1. A *selection function* $\varsigma : H_F \to 2^A$ decides which (non-empty) set of base-line solvers from A is applied to the current problem iteration.
2. A *composition function* $\kappa : H_F \to RES \times REQ$ composes the overall results of the selected base-line solvers and generates further requests.

For steady state estimation, a selection function ς could select all estimators that have not yet detected the end of the warm-up phase. Likewise, a composition function κ could request further data points until the majority of estimators has detected an end of the warm-up phase, and then average their estimates to calculate the overall result.

The actual state space of the synthetic problem solver consists of the feature history and the base-line solver states, and is, consequently, defined by:

$$\mathbb{S}_{synth} = \{(as, res, req) | as \in \mathbb{H}_F \times \prod_{a \in A} AS_a, res \in RES, req \in REQ\} \tag{5.11}$$

Each base-line solver could have a similar state, i.e., synthetic problem solvers can be nested.

5.3.3.2. Implementation

The components described before, allow to define a synthetic problem solver, implementing the general problem solver interface and realizing its `solve` function (see Equation 5.6). The pseudo-code for such a function is given in Figure 2.

It starts with extracting the set of problem features E_P (see line 6, Figure 2). Note that the pseudo-code assumes feature extractors (elements from $\mathcal{F}_P$ and $\mathcal{F}_S$, respectively) to be unique per extraction function, so that they can be collected in a set. Based on the extracted problem features (including the problem features of previous iterations stored in the feature history), suitable problem solvers are selected by the selection function ς (line 9). Afterward, the `solve` method of the selected problem solvers is applied to retrieve their successor states (line 12). The states of the problem solvers that have not been selected in the current iteration remain unchanged. State features are extracted, from the successor states, resulting in the set of extracted state features E_S (line 15). Problem and state features are appended to the feature history H_F, leading to a successor feature history H_F' (line 18) which is used for composing the next result and request of the SPS, by applying the composition function κ (line 21). Result, request, and feature history are, finally, used to create the successor state s' of the SPS (line 24). Figure 5.12 shows the data flows during an execution of `solve`.

Concrete implementations of the synthetic problem solver could be further optimized. For example, algorithm state feature extraction is only necessary if the state has been changed. Otherwise, previously extracted state features can simply be copied. Furthermore, feature extraction and the application of base-line solvers can be parallelized.

To make the SPS as flexible as possible, its elements are integrated as plugins. Regarding composition and selection function, this allows the integration of various mechanisms (e.g., decision trees [151] or neural nets [20]).

Feature Extractors As described in Section 5.3.3, two types of feature extractors are required for the synthetic problem solver: problem feature extractors and state feature extractors. The original feature extractors of SASF are working on model instances. This is not sufficient for an SPS, as

Algorithm 2 Pseudocode for the *solve* function of a synthetic problem solver. The dot-notation as in $s.as.H_f$ (line 9) refers to certain sub-elements of a tuple, in this case the feature history H_F, which is part of the synthetic problem solver algorithm state information as (cf. Equation 5.11, p. 86). Note, that the pseudocode corresponds to the `solve` method of the problem solver interface (see Equation 5.6), and that the SPS is driven by the process of managing a problem solver (see Figure 5.9).

```
Given: synthetic problem solver $SPS = \langle A, \mathcal{F}_P, \mathcal{F}_S, \kappa, \varsigma \rangle$
Input: problem $p^i_{it} \in \mathbb{P}_{it}$, input state $s^i \in \mathbb{S}_{synth}$ (where $i$ is the current iteration)
Output: output state $s^{i+1} \in \mathbb{S}_{synth}$

 //extract problem features
 $E_P \leftarrow \bigcup_{F_P \in \mathcal{F}_P} F_P(p^i_{it})$

 //select baseline solvers
 $Sel \leftarrow \varsigma(s^i.as.H_F, E_P)$

 //apply selected baseline solvers and retrieve successor states
 $Suc \leftarrow \{a.solve(p^i_{it}, s^i.as.s_a) \mid a \in Sel\} \cup \{s^i.as.s_a \mid a \in A \backslash Sel\}$

 //extract state features from successor states
 $E_S \leftarrow \bigcup_{F_S \in \mathcal{F}_S, s \in Suc} F_S(s)$

 //append current iteration $(i+1)$ to feature history
 $H'_F \leftarrow (s^i.as.H_F, (E_P, E_S))$

 //compose next iteration
 $(res, req) \leftarrow \kappa(H'_F)$

 //construct new algorithm state
 $s^{i+1} \leftarrow ((H'_F, Suc), res, req)$
```

feature extractors have to work on problem solver states and on different problem types. Hence, for the SASF extension the `IFeatureExtractor` interface is extended with a generic type `T` that defines on which kind of entity the feature extractor operates. By setting `T` to JAMES II model interface, the corresponding feature extractor works like a standard feature extractor of original SASF on model instances, whereas `T` represents a list of double values could work on a time series that can be regarded as input problem of steady state estimators. Similarly, if `T` is replaced with the class or interface of a problem solver state, the corresponding feature extractor would work on the problem solver state and hence be a state feature extractor. With this generic approach feature extractors get the required flexibility for reasoning on arbitrary algorithms and problems.

As the performance of problem solvers can be considered as features, it is desirable, that performance measurers can be used for feature extraction. Therefore, GUISE offers an abstract feature extractor, which receives factory and parameters of a performance measurer, and returns the measurer's result as feature.

Similar to feature extractors, the `IPerformanceMeasurer` interface has been extended with a generic type `T` representing the kind of solver state it works on. Note that performance measurers do not work on the algorithms as in the original SASF, but on algorithms states, to track the different iterations of problem solving.

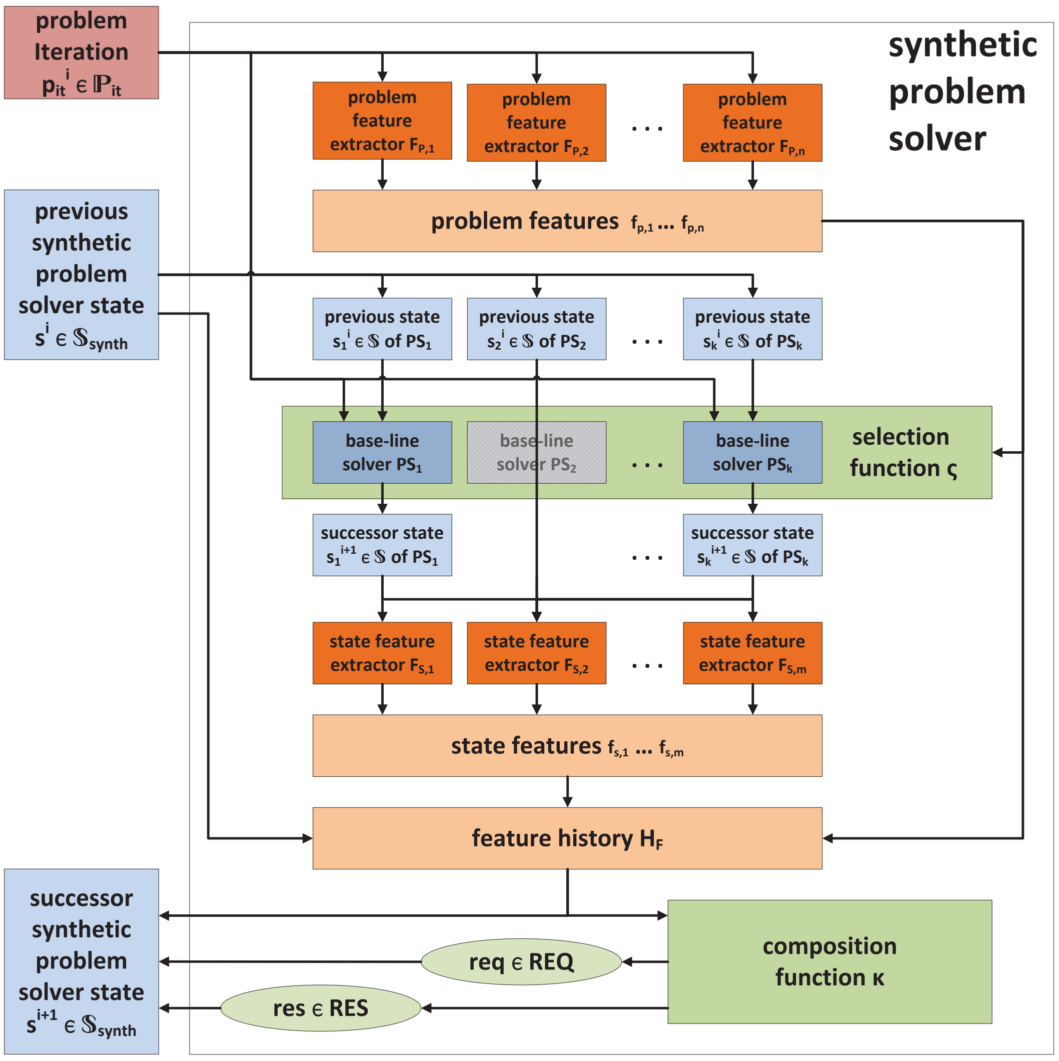

problem Iteration $p_{it}^{i} \in \mathbb{P}_{it}$
synthetic problem solver
problem feature extractor $F_{P,1}$
problem feature extractor $F_{P,2}$
problem feature extractor $F_{P,n}$
problem features $f_{p,1} \dots f_{p,n}$
previous synthetic problem solver state $s^{i} \in \mathbb{S}_{synth}$
previous state $s_1^{i} \in \mathbb{S}$ of PS_1
previous state $s_2^{i} \in \mathbb{S}$ of PS_2
previous state $s_k^{i} \in \mathbb{S}$ of PS_k
base-line solver PS_1
base-line solver PS_2
base-line solver PS_k
selection function ς
successor state $s_1^{i+1} \in \mathbb{S}$ of PS_1
successor state $s_k^{i+1} \in \mathbb{S}$ of PS_k
state feature extractor $F_{S,1}$
state feature extractor $F_{S,2}$
state feature extractor $F_{S,m}$
state features $f_{s,1} \dots f_{s,m}$
feature history H_F
successor synthetic problem solver state $s^{i+1} \in \mathbb{S}_{synth}$
req ∈ REQ
res ∈ RES
composition function κ

Figure 5.12.: The function `solve` of the SPS. It receives a problem iteration $p_{it}^{i} \in \mathbb{P}_{it}$ (Equation 5.7) and the current SPS state $s^{i} \in \mathbb{S}_{synth}$ (Equation 5.11). At first, the problem feature extractors $F_P \in \mathcal{F}_P$ extract relevant features from p_{it}^{i} (cf. line 6 in Figure 2). Then, the selection function ς selects suitable problem solvers for the current problem iteration, p_{it}^{i} (cf. line 9). The selected solvers operate on their previous states $s_1^{i}, \dots, s_k^{i} \in \mathbb{S}$ and are applied to p_{it}^{i}, which results in successor states $s_1^{i+1}..., s_k^{i+1} \in \mathbb{S}$ (l. 12). They are stored in the SPS' successor state s^{i+1} (l. 24), for the next iteration of `solve`. Furthermore, they are forwarded to the state feature extractors $F_S \in \mathcal{F}_S$ (l. 15). Note that state feature extractors may only be applicable to certain problem solver states. For instance, some steady state estimator could store a specific test statistic in its state. A corresponding feature extractor would first check compatibility, and then either extract the statistic or return no feature. The features of the current iteration are appended to the feature history H_F (cf. l. 18). The updated feature history H_F' is forwarded to the composition function κ that creates a result of the current iteration ($res \in RES$), and a request for further information about the problem ($req \in REQ$, cf. l. 22). Finally, result and request are stored in the SPS successor state (cf. l. 24), which is returned and can be used to compute the next iteration of `solve`.

5.3.3.3. An Example *SPS*

An example SPS for steady state estimation shall be outlined in the following.

As mentioned before, simulation-based steady state estimation works on time series. In each iteration, an additional part of a time series is created by simulation and investigated by the estimator, i.e., $\mathbb{P}_{it} = \mathbb{R}^n$. The result is an estimate of some steady state statistic, in this case the steady state mean. Therefore, $RES = \mathbb{R} \cup \{\perp\}$ and $\perp$ again denotes that no steady state mean could be estimated. The request is a boolean value denoting whether additional data are required for estimation, hence $REQ = \{true, false\}$.

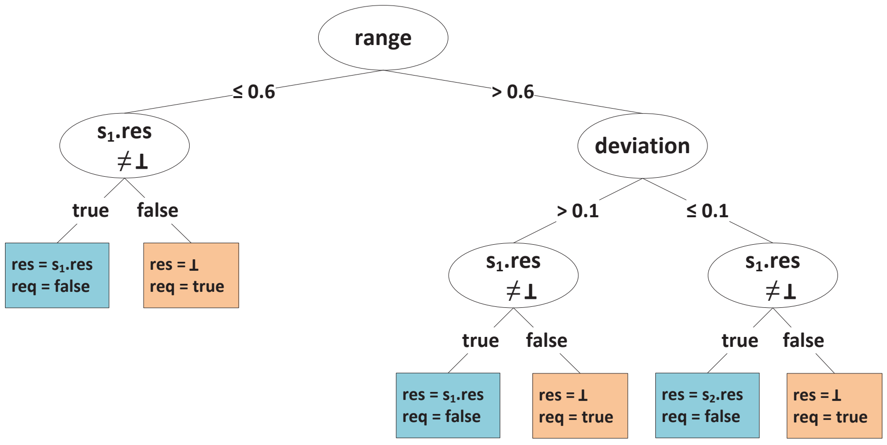


Figure 5.13.: Example decision tree representing the function κ. It is traversed from top to bottom until a leaf is reached, and works on both problem and state features. At first, the (relative) range is considered: if it is below or equal to 0.6, the left sub-tree is selected, otherwise the right one. In the left sub-tree, a decision is made depending on the state feature of estimator a_1, by determining whether it could estimate a steady state mean, stored in its state, i.e., $s_1.res \neq \perp$. In the right sub-tree, the standard deviation is requested. If the deviation falls below 0.1, the state feature of estimator a_1 is considered, otherwise that of estimator a_2 is used. The leaves of the tree denote result and request of the SPS instance. Depending on problem and state features, κ either requests more data points from the simulation or returns the result of the algorithm that is considered more suitable for trajectories with the given features (see line 22 in Figure 2)

.

The sample $SPS = \langle A, \mathcal{F}_P, \mathcal{F}_S, \varsigma, \kappa \rangle$ uses two base-line problem solvers, i.e., $A = \{a_1, a_2\}$, where a_1 is the MSER steady state estimator [190] and a_2 is Schruben's test steady state estimator [166].[3] $\mathcal{F}_P$ contains two problem feature extractors, one for the standard deviation and one for the range, i.e., the distance between minimum and maximum value, of the time series. $\mathcal{F}_S$ contains a state feature extractor that retrieves the estimated steady state mean from the algorithms in A, stored in their states, which may be $\perp$ to denote that none was found yet.

The selection function ς is trivial, it always selects both algorithms: $\varsigma(H_F) = \{a_1, a_2\}$. The composition function κ is based on a decision tree (e.g., [151]), which is generated from previously collected performance data, where a_1 and a_2 have been applied to representative problems. Figure 5.13 shows a hypothetical decision tree that could result from this training process.

[3]This is a very brief example; more realistic case studies are presented in chapters 6 and 7.

5.3.3.4. Realizing Existing Composition Approaches

As one major objective of the SASF extension is flexibility regarding the used selection & composition mechanism, the following paragraphs show how an *SPS* instance can be configured to realize algorithm selection, algorithm portfolios, algorithm ensembles, and online adaptation of algorithms. Each paragraph also discusses possible applications.

Mapping the Algorithm Selection Problem As described in Section 5.1.1, algorithm selection extracts features $f_x \in \mathbb{F}$ from a problem $x \in \mathbb{P}$ to select among a set of algorithms A the most suitable for x. Selection is done by a selection mapping S that is generated by considering performance measures $p(a, x) \in \mathbb{R}^n$ and user criteria $w \in \mathbb{R}^k$.

The first element of an $SPS = \langle A, \mathcal{F}_P, \mathcal{F}_S, \varsigma, \kappa \rangle$, A, corresponds to the set of algorithms A in the algorithm selection problem. However, each algorithm has to comply with the problem solver interface (see Equation 5.6). The problem feature extractors in $\mathcal{F}_P$ are analogous to feature extractors in the ASP, but $\mathcal{F}_P$ would also include a problem feature extractor that returns the user criteria $w \in \mathbb{R}^k$. The set of state feature extractors, $\mathcal{F}_S$, contains a single extractor that retrieves the overall results from the state of the selected base-line problem solver. As only one base-line problem solver shall be selected and used, the selection function ς picks a single element from A and thus realizes the selection mapping S of the ASP (see Equation 5.3). Typically, ς is generated by analyzing algorithm performance on training problems. This analysis relies, at least implicitly, on some measure of performance, which is explicitly considered by the ASP but has no direct counterpart in the *SPS*. The composition function κ returns the results extracted from the state of the selected base-line problem solver. Also note that the ASP problem space $\mathbb{P}$ is a special case of the iterated problem set $\mathbb{P}_{it}$ (Equation 5.7) — each element comprises an empty history.

As a sample setup, consider a set A of *Stochastic Simulation Algorithms (SSAs)* for chemical reaction networks [68]. SSA performance strongly depends on the given model, so that automatically selecting a suitable algorithm can considerably improve execution time [48]. The problem space $\mathbb{P}_{it}$ contains all chemical reaction networks that can be simulated with SSAs. Problem feature extractors in $\mathcal{F}_P$ may retrieve the number of distinct species and reactions from a model, or a measure of its stiffness. An additional user criterion could, for example, account for approximative SSA variants and specify an acceptable trade-off between execution speed and accuracy. The state feature extractor in $\mathcal{F}_S$ would retrieve the generated simulation trajectory from an SSA state, and the selection function ς could be generated via supervised learning on previously recorded execution times. A similar approach was pursued in [104] for spatial SSAs. The composition function κ would return the generated simulation trajectory to the user.

Mapping Algorithm Portfolios Instead of selecting a single base-line algorithm, the fundamental idea of algorithm portfolios is to apply a set $A \subseteq \mathbb{A}$ of algorithms to the same problem and to combine their results, thereafter. This approach originates in finance, where financial assets are bundled into portfolios that maximize the expected return, given the risk aversion preferences of an investor [128]. For algorithms $a_1, \ldots, a_n \in \mathbb{A}$ one could define a weight vector determining to which degree an algorithm belongs to the portfolio:

$$\vec{\alpha} = (\alpha_1, \ldots, \alpha_n) \in [0,1]^n, \text{with} \sum_{i=1}^{n} \alpha_i = 1 \tag{5.12}$$

If an $SPS = \langle A, \mathcal{F}_P, \mathcal{F}_S, \varsigma, \kappa \rangle$ realizes an algorithm portfolio, the non-zero elements of the portfolio's weight vector $\vec{\alpha}$ (see Equation 5.12) determine the algorithm set A. Problem features are typically neglected for portfolio construction, so $\mathcal{F}_P$ would be empty. $\mathcal{F}_S$ would contain at least two state feature extractors: one to retrieve the current solution from a base-line problem solver state, and one to retrieve its requests. The selection function ς would select all portfolio members for execution, whereas the composition function κ would aggregate the base-line problem solver's results and requests based on the weight vector $\vec{\alpha}$, i.e., it corresponds to the function f in Equation 5.12.

Portfolios have been successfully applied to SAT solving (e.g., [196]) and other NP-hard problems (e.g., [95, 72]), as well as the simulation of chemical reaction networks in JAMES II [48]. Portfolio-based approaches receive attention in many other domains, e.g., for valuating and managing biodiversity [55], and for exploring biological robustness and a suitable balance of fragility and performance in cell biological systems [114].

In case of simulation-based optimization, each base-line problem solver could be a parameter search method that requests new points in a model's parameter space to be evaluated. The κ function could aggregate all requests based on portfolio weights, e.g., to let more promising optimization algorithms evaluate more data points. Dynamic portfolios, where the weight vector $\vec{\alpha}$ is adapted during iterations, can be mapped to an online adaptation scheme (see below).

Mapping Algorithm Ensembles Algorithm ensembles are predominantly used in machine learning, where learning algorithms are applied in two steps: first, to train algorithms $a_i \in \mathbb{A}$ to (subsets of) input data and, second, to train a function that operates on top of them. Hence, in contrast to portfolios, algorithm ensembles are not defined by a real-valued vector $\vec{\alpha}$ (see Equation 5.12), but instead introduce an additional function g that considers the results of algorithms $a_i \in \mathbb{A}$ and acts upon them:

$$g : \mathbb{P} \times \mathbb{A}^n \to RES, \tag{5.13}$$

where RES is the set of possible results, algorithms $a_i \in \mathbb{A}$ can produce.

The realization of ensembles by the SPS is quite similar to algorithm portfolios. The only difference is that the composition function κ now realizes an ensemble function that takes all solver results into account. This may also require that the original problem $x \in \mathbb{P}$ is stored as a problem feature so that it can be accessed by g (see Equation 5.13).

Algorithm ensembles are predominantly used in machine learning, where various methods exist to train the $a_i \in \mathbb{A}$ to (subsets of) the input data first, and to train a function κ that operates on top of them, afterwards. κ may either realize a portfolio approach, i.e., simply aggregate the result by algorithm-specific weights, or it may learn to decide which base-line algorithm a_i performs well in which region of the problem space $\mathbb{P}_{it}$, i.e., which result to choose in which situation [157, 198]. In machine learning, the latter approach is called meta-learning; it can be tackled with methods to solve the ASP [169]. The approach of learning κ from empirical data is promising, since the results of the base-line problem solvers in A may reveal important properties of the problem at hand, and can be extracted by state feature extractors from $\mathcal{F}_S$. Ensemble methods are widely used in bioinformatics, e.g., for proteomics data analysis [198], and the case studies in the following two chapters 6 and 7 rely on ensemble approaches.

Online Adaptation All of the previously discussed schemes can be adapted to work in an online fashion, leading to dynamic algorithm selection, dynamic algorithm portfolios, and dynamic algorithm ensembles. In general, adaptive behavior in the SPS can be realized by defining problem and state feature extractors that extract metrics regarding the solution progress. Such data is then stored to the feature history H_F, so that the ς and κ functions can access them. Data to be accessed by the solvers itself can be added to the answer-request history by the composition function κ, so that it becomes part of the next problem iteration $p \in \mathbb{P}_{it}$.

Approaches that adaptively reconfigure a simulator at runtime (e.g., [134, 79]) can be mapped to this scheme as well. The selection function ς would realize the online learning algorithm and select the most suitable simulator for the next part of the simulation task, while the composition function κ would simply store the last model state (extracted from the selected base-line problem solver) to the answer-request history, so that it can be accessed by the next base-line problem solver. Reinforcement learning can also be used to adapt simulation sub-tasks like load balancing at runtime (e.g., [134]). A mechanism that has been successfully applied to the simulation of chemical reaction networks and ML-Rules models in JAMES II, relies on reinforcement learning to adaptively re-configure simulators [79].

Another example for such an approach is Adaptive Online Time Allocation (AOTA) [62]. It adapts the weights of search algorithms according to their performance observed so far, and works on a sequence of problems $(p_1, ..., p_m) \in \mathbb{P}$ as well as a set of algorithms $\mathbb{A}$ applicable to the problems in $\mathbb{P}$. Let $t_{i,j}$ be the time an algorithm $a_i \in \mathbb{A}$ spent on solving a problem $x_j \in \mathbb{P}$, and let $f_{i,j} \in \mathbb{F}$ be the *current* features of problem x_j as it is being solved by a_i. A history of problems $x_1, \ldots, x_k \in \mathbb{P}$ can be defined as:

$$H = \bigcup_{j=1}^{k} H_j, \text{with } H_j = \{(f_{i,j}^{(r)}, t_{i,j}^{(r)}) | r = 0, ..., h_{i,j}\} \tag{5.14}$$

where $h_{i,j}$ is the number of iterations that algorithm a_i needed to solve problem x_j [62]. Hence, the history consists of tuples with time-stamped features that describe the iterative solution processes of the algorithms on the past problems. The goal is now to find a function g that predicts the *additional* computing time an algorithm a_i needs until it solves a problem x_j. The prediction is based on the tuple of the current round r, $(g_{i,j}^{(r)}, t_{i,j}^{(r)})$, as well as the history H (see eq. 5.14). Predictions of g, which should become more accurate with a growing history H, are used to allocate computing resources to the algorithms in the following round.

The AOTA history H can be mapped to the feature history H_F of the synthetic problem solver. AOTA's online adaptation could be realized by letting the selection function ς select all algorithms in A for execution, and by letting the composition function κ store the algorithm-specific predictions (i.e., how fast it will solve the given problem) as part of the results (RES). Each base-line problem solver could access these results during the next iteration (since they are part of the iterated problem $p \in \mathbb{P}_{it}$) and let the prediction result determine its resource usage (see [62] for details).

This approach allows to learn portfolios incrementally and lends itself to an integration of various learning methods, e.g., reinforcement learning [172].

5.3.4. Generating Synthetic Problem Solvers

The SPS is configured and some of its components are generated in a process that is described in the following, and depicted in Figure fig:wf:spdm. Initially, a performance experiment is executed

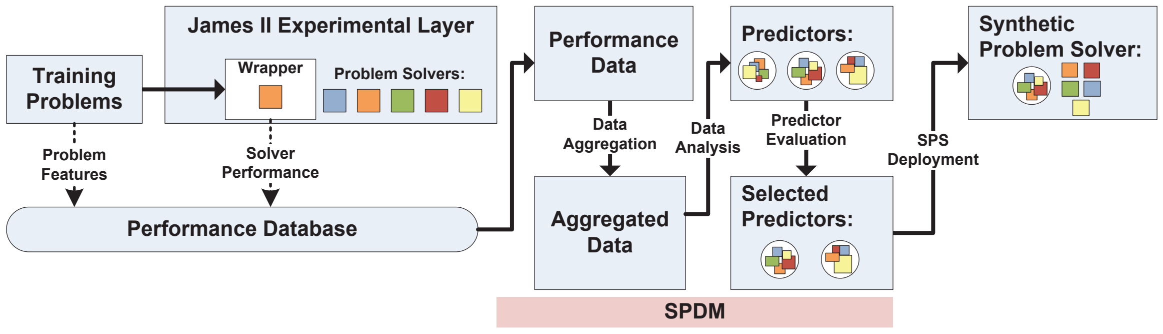


Figure 5.14.: SPS generation via the SPDM. The SPDM-based processing steps (marked by a red bar at the bottom) rely on problem features and solver performance measurements from the performance database. Performance data are aggregated and analyzed. Eventually, this yields predictors that can then be used within a synthetic problem solver.

with a set of given training data. In SPDM performance experiments can be easily set up by using JAMES II's experimentation layer (see Chapter 3, pp. 29) which applies simulation algorithms to models and collects performance data through observation. To reuse this mechanism, a wrapper has been implemented, that mimics a simulator and applies a problem solver to a given problem. JAMES II' experimentation layer can be used to set up large-scale performance evaluations [83].

Afterward, the results of the performance experiment are automatically stored in the performance database, and further processed. The subsequent steps, as well as the infrastructure for generating training data and storing performance data, are described in the following.

Generating Training Problems To train the composition component of the SPS, training data have to be created. A set of training data TD is usually a sub-set of the set of iterated problems $\mathbb{P}_{it}$. The main challenge of generating a set of training data is to make it representative, i.e., as $\mathbb{P}_{it}$ is often very large or even infinite, the training set TD should be as small as possible, while its elements cover, ideally, all relevant characteristics of all problem instances $p \in \mathbb{P}_{it}$.

To facilitate the integration of algorithms that generate such training sets, the `IProblemGenerator` interface is introduced. Objects implementing this interface are instantiated with a set of parameters, to define the characteristics of the problem to be generated (e.g., size, bias, etc. of a time series). For creating problems, the interface offers the method `generate`, which receives a random seed and returns a problem instance. The seed is responsible for generating stochastic effects in the generated problem instance, leading to different instances for one problem parametrization. This corresponds to the problem structure of the Spdm performance database (see Section 5.1.4), where model description and model parameters are distinguished from model instances. The concept is now generalized for arbitrary problems. A problem generator is identified by an URI similar to the model description, it receives a set of problem parameters similar to the model parameters, and parametrized problem generators create problem instances based on a random seed, similar to a model instance.

This approach is easily applicable to different problem descriptions, e.g., for time series, which are input problems for steady state estimators, a problem generator can be defined based on a set of parameters, like time series length, initial bias length, etc. As time series usually include stochastic noise, different instances of a time series with the same characteristics can be generated using different seeds.

Adaptation of the Performance Database The performance database (see Section 5.1.4) of Spdm stores performance data of simulation algorithms, for analysis and prediction. As described, its design is oriented on the ASP. For extending the performance database, the focus on problem models and simulation algorithms has to be generalized, however, most of the the original database's entities can be reused. The existing infrastructure can be reused to store problem features, whereas a storing of state features is not required, as state features rely on performance measurers (see 5.3.3.2), whose results are similarly handled as in the original performance database. The entities for storing algorithm are reused as well, as they represent parameter configurations that can be used for arbitrary problem solvers.

The only entity which needs adaption is the problem scheme, which now represents general problems, not only simulation models. However, problem generator descriptions are similarly structured as model descriptions, i.e., they are described by a location URI and require parameters to be instantiated. The only change necessary at this point is to store general problem definitions (e.g., locations) instead of model descriptions.

Data aggregation In some cases performance data have to be aggregated before training the selection & composition mechanism. This is the case, e.g., if problem solver applications are replicated and the mean of their performance results has to be calculated, or if the performance of a problem solver (e.g., the estimated steady state mean) has to be set into relation to the desired performance (e.g., the real steady state mean). For this, an abstract data aggregator component has been designed, which works on a given type of performance result. Aggregators are integrated as plugins, making a custom data aggregation, tailored for specific performance measures, possible

Analyzing Performance Data to Generate Performance Predictors Aggregated performance data are analyzed to generate a selection & composition mechanism. SASF's mechanisms to generate performance predictors is reused and extended for this task. To realize selection & composition (and not only algorithm selection), the `IPerfComparisonPredictor` interface (see Section 5.1.5) is adapted. Many tools exist that offer functionality for this interface, as they provide different classification and prediction algorithms, e.g., Weka [77], MLJ [94] or Joone [129].

The adapted interface offers the method `predictPerformance`, which receives a runtime configuration and a set of features, and returns a double value representing the suitability class of the runtime configuration (i.e., problem solver) for the given features. Selection & composition mechanisms implementing the performance predictor interface can be used by the default implementation of the SPS' composition function κ.

For instance, a performance predictor could be based on a decision tree, where each branch represents a feature and each edge represents a feature value. Leaves would comprise the runtime configurations of those algorithms that are considered to be most suitable for the feature values that determine the path to the leaf (an example for this is the decision tree shown in Figure 5.13, p. 89). The predictor would return 1.0, if the given features lead to the given runtime configuration; otherwise it would return 0.0. The composition κ retrieves the performance predictions for all algorithms and takes the one with the the highest result (1.0 in the example) for generating result and request (see Section 5.3.3.2).

Predictor Evaluation Finally, the predictors generated by the Spdm are evaluated with methods like bootstrapping or cross-validation. While this last step is important to check whether the method used for data analysis is suitable, e.g., the prediction error is low (Spdm supports custom error metrics), it should be noted that this does *not* necessarily mean that the resulting SPS instance is suitable for arbitrary problems of the corresponding problem space. The effectiveness of SPS instances has to be thoroughly evaluated in additional experiments.

Deploying Synthetic Problem Solvers to the Algorithm Selection Registry The ability to apply selectors (see Section 5.1.6) that are tailored to selecting simulation algorithms, is not sufficient for deploying synthetic problem solvers. Generated synthetic problem solvers need to be accessible via the same mechanisms as normal plugins, so that users can use them as if they were standard plugins. To automate SPS deployment, their factories and plugin descriptions have to be generated and integrated automatically, as well.

As SPS instances may not indicate their application domain, additional data that is important for selecting among them, has to be provided. For instance, an SPS instance for steady state estimation could have a similar input output behavior as one for cycle detection in time series, but pursues a completely different goal. Hence, to select the right SPS instance, at least *one* specific JAMES II factory is created for each plugin type that supports synthetic problem solvers (e.g., one could build an own factory for steady state estimators, and one for cycle detectors, which are now distinguishable, even though both belong to the single run analysis plugin type). These factories implement the interface, `ISyntheticPluginFactory` and get access to the list of all available synthetic plugins for their plugin type, and return properly configured synthetic problem solvers when their factory methods are called.

The overall structure of the newly developed types for the JAMES II registry is sketched in Figure 5.15. The algorithm selection registry has been furthermore extended to handle SPS plugins. Instead of plugin XML files, new plugin descriptions are required, as different SPS implementations for the same plugin type might rely on the same factory. In addition, meta-information that is not stored in plugin XML files, e.g., the kind of used training or evaluation data, might be required. Therefore, plugin descriptions are wrapped into an additional components — *synthetic plugin* objects, which are maintained by an auxiliary management component.

The synthetic plugin objects are stored in a local file which is loaded during start-up of the registry. Despite this storing in an own data structure (which is separated from other JAMES II plugins), calls for plugins of a certain plugin type consider all synthetic plugins of that type. Hence, the user can treat synthetic plugins the same way as standard versions.

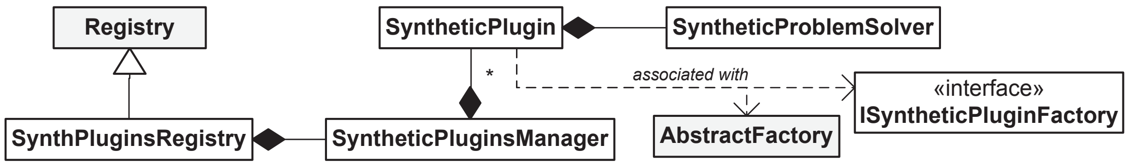


Figure 5.15.: The algorithm selection registry is extended to handle synthetic plugins. The registry inherits from the original registry (top-left corner, gray) and delegates plug-in requests to the `SyntheticPluginsManager`, which manages all synthetic plugins (e.g., w.r.t. their persistence). Each synthetic plugin belongs to a plugin type that is denoted by the corresponding `AbstractFactory` (bottom middle, gray) in JAMES II, and contains an instance of the synthetic problem solver (see Section 5.3.3), some meta-data (e.g., date of deployment), as well as a reference to the specific synthetic plugin factory which initializes instances of the synthetic problem solver.

5.4. Summary

This chapter described the integration of algorithm selection & composition mechanisms into the experiment structure. Therefore, a general problem solver interface has been introduced that conforms to algorithms applied during the six tasks of a simulation experiment (see Section 2.3.1, pp. 25). The interface is used by the workflow described in Chapter 4 to integrate such algorithms. SASF, a framework for algorithm selection on simulation algorithms, has been extended, to realize algorithm selection & composition.

Based on the problem solver interface, a synthetic problem solver has been derived that considers features from problem and algorithms to compose improved solutions. It can incorporate different composition strategies, e.g., algorithm selection, portfolio, or ensemble learning. The algorithm selection registry of SASF has been extended for deployment and seamless integration of SPS definitions as plugins, and SASF's capabilities have been extended to train and generate an SPS.

The next two chapters, 6 and 7, will show the applicability of the SPS for steady state estimation algorithms and statistical analyzers for estimating the number of replications.

Nevertheless, the process of generating SPS instances could be improved. While all required tools for creating an SPS are provided, the development of a workflow for an automated SPS generation could be beneficial. To realize that, techniques taken from the workflow management system WorMS (see Chapter 4.1) might be used. In addition, a GUI should guide users through the process of generating an synthetic problem solvers, i.e., to let users create performance data, based on input problems with features that are relevant to them.

6. Case Study I: Synthesizing a Steady State Estimator

"All biologic phenomena act to adjust: there are no biologic actions other than adjustments. Adjustment is another name for Equilibrium. Equilibrium is the Universal, or that which has nothing external to derange it."

Charles Fort

Steady state estimation is a non-trivial issue in many scientific fields, e.g., chemistry [9], biology [26], or economics [36], and is of special interest for simulation experiments, as it represents a typical (sub-) goal for the single-run analysis, i.e., the analysis of a simulation run's output data. A steady state estimator tries to find the first point in a time series, where the warm-up phase is finished and calculates a statistic (e.g., the mean) for the remaining points. Assmussen et. al. [8] proved that no universal solution exists for finding that point, i.e., no solution *always* yields correct results (independently of specific trajectory features). However, it is still possible to construct a solution that performs particularly well for *certain* kinds of problems, e.g., those that are particularly relevant in practice. Hence, steady state estimation is an adequate example for illustrating the benefits of the SASF extension presented in Chapter 5.

The aim of this chapter is to join several *base-line* problem solvers for steady state estimation to a larger, more robust SPS instance (see Section 5.3.3). Therefore, various well-known steady state estimators are applied to a set of generated problems and their performance is analyzed. As the approach relies on a broad and representative set of sample data, a problem generator to produce synthetic problem time series is used. The parameters of the problem generator are selected in order to create time series that cover relevant characteristics for steady state mean estimation.

The experiment results are stored in the adapted performance database and used to train an SPS instance for steady state estimation. This synthetic steady state estimator is described by its elements, and deployed to the extended algorithm selection registry (see Section 5.3.4). Its advantages are shown in an evaluation experiment with the training data, as well as simulation output data generated by biological and chemical models.

6.1. Evaluation of Steady State Estimators

To show, that different steady state estimators vary in their suitability for estimating a steady state statistic on different problem series, an evaluation experiment with ten different steady state estimators is presented. To illustrate the experiment, the estimators, as well as the method for problem data generation are described. While the estimators work iteratively, they are applied just once, i.e., not iteratively, on each problem, as their ability to detect the end of the warm-up phase in a time series chunk can be most easily evaluated by confronting them with fixed time series, representing such chunks. Evaluation results are stored in the performance database (see sections 5.1.4, pp. 72 and 5.3.4, p. 93) and discussed at the end of this section.

6.1.1. Steady State Estimation in Simulation

Various definitions for the term *steady state* exist. An informal definition is given by the American Heritage Science Dictionary, where a steady state is defined as *"a condition of a physical system or device that does not change over time, or in which any one change is continually balanced by another, such as the stable condition of a system in equilibrium"* [78, p. 593]. This represents a very intuitive idea of the term. Nevertheless, for an algorithmic treatment, a formalization is required, which is given, e.g., by Assmussen and Glynn [7, p. 96]: Given a continuous, stochastic process $S = \{S(x)\}_{x \geq 0}$, the steady state is a constant $\widetilde{S}$ such that

$$\frac{1}{t}\int_0^t S(x)dx \overset{p}{\to} \widetilde{S}, \tag{6.1}$$

where $\overset{p}{\to}$ denotes convergence in probability and $t \to \infty$.

This way of defining steady states is, however, not suitable for cases where the stochastic process S is represented by a simulation model. If such a model is complex, it is typically difficult to solve the problem (i.e., to find a steady state) analytically. Instead, a steady state statistic can be *estimated* from simulation output. As this statistic is, usually, the steady state mean, the term *steady state estimation* will refer to *steady state mean estimation*, in the following. An obvious estimator for calculating the steady state mean $\hat{S}$[1] is given by Assmussen and Glynn as well [7, p. 97]:

$$\hat{S} = \frac{1}{t}\int_0^t S(x)dx \tag{6.2}$$

It uses one sample of size t for estimation, which comprises infinitesimal many points.

Simulation output trajectories, however, are discrete, i.e., they do not comprise infinitesimal many time steps. Hence, the integral used in Equation 6.2 is not necessary, in this case. A trajectory can be easily transformed into a time series (comprising equidistant instead of varying time steps) by interpolation. A possible steady state estimator for the resulting time series, operating on the discrete time interval $[1, t]$ would be:

$$\hat{S} = \frac{1}{t}\sum_{i=1}^{t} S(i), \tag{6.3}$$

Many simulation runs start with a so-called *warm-up phase*, which hampers the estimation of steady state means by the above estimator because it introduces an *initial bias*. This part of the simulation output trajectory is also called the *initial transient*. A reliable steady state estimator has to *detect* the point at which the warm-up phase is over. As many ways to do so exist, a variety of approaches have been developed (e.g., [45, 193, 166, 144, 27, 190, 111]), all of which estimate the steady state mean based on simulation output and only differ in their mechanism to detect the end of the warm-up phase. This is illustrated by the following working definition of a *Steady State Estimator (SSE)*, which incorporates a detector D_B for the end of the initial bias:

$$D_B : \mathbb{R}^n \to \mathbb{N} \tag{6.4}$$

Given simulation output $\vec{y} = y_1, ..., y_t$ (where $y_i \in \mathbb{R} \wedge t \in \mathbb{N}$), it determines at which point the initial bias of the simulation is over. An SSE is a function:

$$SSE : \mathbb{R}^t \to \mathbb{R} \cup \{\perp\}, \tag{6.5}$$

$$SSE(\vec{y}) = \begin{cases} \frac{1}{t-D_S(\vec{y})} \sum\limits_{i=D_B(\vec{y})}^{t} y_i, & \text{if } D_B(\vec{y}) < t \\ \perp, & \text{otherwise} \end{cases} \tag{6.6}$$

[1]Note that a different symbol $\hat{S}$ is used for the estimated steady state mean, to denote the difference to the true steady state $\widetilde{S}$ of Equation 6.1.

As t denotes the end of the time series, steady state mean estimation is only possible if the detected end point of the warm-up phase ($D_B(\vec{y})$) is lower than t. Otherwise, no estimation is possible, which is represented by the *null* element $\perp$. In this case, the estimation has to be continued with additional data points.

6.1.2. Used Steady State Estimators

Ten different steady state estimators have been used for the experiment. They work according to the definition given in Equation 6.6 of Section 6.1.1. They implement the problem solver interface as described in Section 5.2.3 and only differ in the used detector D_B. The following detection strategies have been used:

1. MSER: identifies the warm-up phase's end by deleting initial observations to the point that provides the minimal MSER statistic [190].
2. Euclidean Distance: divides a time series into vectors and normalizes them. If all vectors of a sequence are close enough to the unit vector, the end of the warm-up phase has been detected [120].
3. Goodness of Fit: divides a time series into batches, counts the amount of values below and above the mean for each batch, and performs a Chi-Square test on the resulting histograms to identify the warm-up phase [144].
4. Balancing Mean: counts the amount of values above and below the mean of a time series. If the difference between both counts is below a given threshold, the end of the warm-up phase has been detected [45].
5. Running Mean: assumes the end of the warm-up phase as soon as the change in the running mean falls below a given threshold [144].
6. Batch Mean: divides a time series into batches and the batches into two groups. Assumes the end of the warm-up phase as soon as the distributions of the variances for the two groups are close enough [27].
7. Crossing Mean: counts the crosses of a time series and its running mean. As soon as a given amount of crosses occurred, the end of the warm-up phase is assumed [193].
8. Stop Crossing Mean: counts the crosses of time series and running mean. As soon as no crosses happened for a given length, the end of the warm-up phase is assumed.
9. Schruben's Test: estimates the stationarity of a time series to identify its warm-up phase [166].
10. Moving Windows: moves a window through a time series and updates the mean according to the values inside the window. If the standard deviation falls below a given threshold, the warm-up phase has been detected [111].

Strategies $1-3$ have been proposed by [87] for automated experimentation as they do not require a careful configuration. The others have been proposed during the last four decades and are still in use. This illuminates the difficulty of selecting the most suitable SSE. All estimators implement the problem solver interface introduced in Section 5.2.1 according to the mapping described in Section 5.2.3. The result of each solver is the estimated steady state mean or a bottom element $\perp$ (if no steady state mean could be estimated), hence $RES = \mathbb{R}_0 \cup \{\perp\}$. The request is a boolean value denoting whether additional data points are required for estimation, hence $REQ = \{true, false\}$. Furthermore, the estimators have been realized as plugins of JAMES II, and are used with their default parameters.

6.1.3. Problem Data

For generating the problem data, a problem generator implementing the `IProblemGenerator` interface introduced in Section 5.3.4, pp. 93 is used. It follows the approaches of [27], [171], [191], and [87] of a successive generation of time series, instead of using an M/M/1 queue as, e.g., [60], [61], and [192] do. The reason for this decision is its straightforward parametrization, which allows for a direct control of several key features (see below). The problem generator has the following structure:

$$G = \langle t, s, l_b, l_o, n, c \rangle \tag{6.7}$$

It generates time series with bias of length l_b, trend t, and shape h. The noise of the time series is induced by a random number generator and varies between $-n$ and $+n$. The series is of size l_o and the numbers are cross correlated with factor c.

Trend and shape are used to cover the most relevant types of time series. This selection is oriented on previous work by [27, 171, 191, 87], with a more fine-grained investigation of initial bias by distinguishing between two dimensions. The first dimension is bias trend t which includes two variants: constant and quadratic. The second dimension is bias shape h, where three variants are considered: a straight line shape, an oscillating shape, and additionally a random shape. By combining the different shapes and trends, 6 different types of bias can be created. For instance, a quadratic trend with an oscillating shape means that the initial bias oscillates with a quadratically decreasing amplitude, as shown in Figure 6.1.

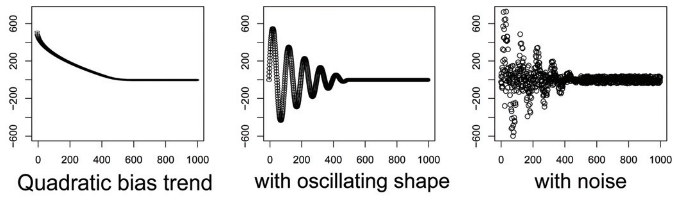


Figure 6.1.: Successive generation of problem time series.

In addition to bias, three other factors have been varied in the experiment: noise, cross-correlation, and bias length. Bias lengths from 0 to 150 percent of the time series length are tested. Bias length of more than 100 percent allows to investigate the ability of an estimator to handle heavily biased time series. Furthermore, as stochasticity plays a role in steady state estimation, the noise level is varied to investigate the robustness of the estimators against them. Noise levels range from 0 to 20 percent, where, e.g., 10 percent noise means that a random value between -10 and 10 percent is added to each value of the time series. The noise has been generated with an auto-correlation factor of 0.5 to get more realistic random numbers that are not i.i.d. Finally, the length of the time series varies between 100 and $1,500$ time points. All of these features are induced on a baseline of 1.0. Hence, if an *SSE* is applied on one of the generated time series the estimated steady state mean should be 1.0. Unfortunately, the constant bias trend with a bias length of at least 100 percent hampers the performance experiment, as it results in a constant (noisy) line that might be far away from 1.0. In this case, it is impossible for an *SSE* to estimate the expected steady state mean of 1.0. Hence, problems with constant bias trend and at least 100 percent bias length are removed from the problem generator parameters.

Table 6.1 shows the used parameters for problem generation. $5,760 - 1440 = 4320$ problem definitions are generated, each being used to randomly generate 10 time series (to reduce stochastic factors) so that the problem data consists of $43,200$ problem instances.

Note that steady state estimators are not applied iteratively on the problem data. They are rather applied once on each generated time series, as this facilitates evaluating their ability to detect the end of the warm-up phase in a fixed time series with corresponding features. Resulting knowledge can then be used by a composition function, in different iterations.

	constant distance in percent	quadratic ascent in percent
Bias trend t	-100 -60 -30 30 60 100	-100 -60 -30 30 60 100
Bias shape s	straight, oscillating, random	
Bias length l_b in percent	0, 50, 100, 150	
Noise n in percent	0, 5, 10, 10, 20	
Autocorrelation factor	-0.5, 0.5	
time series Length l_o	100, 500, 1000, 1,500	

Table 6.1.: Parameters for generating the problem time series.

6.1.4. Performance Criteria

For evaluating the steady state estimators, two performance criteria are used. The first one is the (relative) *deviation* d between the real steady state $\widetilde{S}$ and the steady state mean $\hat{S}$ estimated by a steady state estimator. It gives a measure for the accuracy of the steady state estimator under investigation. The deviation is calculated by:

$$d(SSE, \vec{y}) = \frac{|SSE(\vec{y}) - \widetilde{S}|}{\widetilde{S}} \tag{6.8}$$

Note, that the deviation can only be calculated, if SSE produced a result, i.e., $SSE(\vec{y}) \neq \perp$.

The second performance criterion is the *success rate* of a steady state estimator. It can be regarded as the probability of correctly answering the question whether the warm-up phase has ended within a given time series. A steady state mean estimator SSE shall avoid two kinds of errors: on the one hand, detecting the end of the warm-up phase although it has not ended yet (false positive); on the other hand, *not* detecting the end of the warm-up phase although it has already ended (false negative).

Given a set of time series $\vec{y}^1, \ldots, \vec{y}^m$ and a set of indices $u^1, \ldots, u^m$ denoting the ideal truncation points of the time series (i.e., the end of their warm-up phases), the **success rate** sr of SSE on those time series can be calculated by:

$$sr = \frac{1}{m} \cdot \sum_{i=1}^{m} succ(SSE, \vec{y}^i, w^i), \tag{6.9}$$

with

$$succ(SSE, \vec{y}, u) = \begin{cases} 1, & \text{if } (SSE(\vec{y}) \neq \perp \ \wedge \ u < length(\vec{y})) \\ & \quad \vee \ (SSE(\vec{y}) = \perp \ \wedge \ u \geq length(\vec{y})) \\ 0, & \text{otherwise} \end{cases} \tag{6.10}$$

The function *succ* in Equation 6.10 identifies whether a steady state estimator SSE is successfully applied to a time series. The application is considered to be successful, if the steady state estimator found the end of the warm-up phase (i.e., $SSE(\vec{y}) \neq \perp$) and the ideal truncation point for $\vec{y}$ is smaller than the length of $\vec{y}$ (i.e., $u < length(\vec{y})$), or if the steady state estimator did not find the end of the warm-up phase (i.e., $SSE(\vec{y}) = \perp$) and the index of the ideal truncation is greater or equal than the size of time series (i.e., $u \geq length(\vec{y})$). Otherwise, the estimator is considered to be unsuccessful. The rate of false negatives ($SSE(\vec{y}) = \perp \wedge u < length(\vec{y})$) and false positives ($SSE(\vec{y}) \neq \perp \wedge u \geq length(\vec{y})$) can be defined accordingly.

The ideal truncation point w^i for time series $\vec{y}^i$ is calculated by using the problem generator G^i which generated $\vec{y}^i$. The generator is used to generate a set of sample time series. Wilson and Pritsker [192] proposed an algorithm for getting the ideal truncation point of a set of time series created by the same generator. It is applied on the generated time series and calculates the minimal *Mean Squared Error (MSE)* between each time series data points and real steady state state.

As the steady estimators are implementing the problem solver interface, their results have to be extracted from their state objects. Hence, for getting the result of SSE $(\vec{y})$ in Equation 6.10 a performance measurer p_d is used, which works on the output state s of an SSE:

$$p_d(s) = \begin{cases} 1, & \text{if } s.req = false \\ 0, & \text{otherwise} \end{cases} \tag{6.11}$$

It can be used to store results directly in the performance database and can be reused as feature extractor by the SPS instance.

6.1.5. Evaluation Results

Figure 6.2 shows the mean deviations and success rates of the investigated steady state estimators applied on the problem data described in Section 6.1.3 (remember that the estimators are applied just once, i.e., not iteratively, on each time series). It is visible that the used steady state estimators

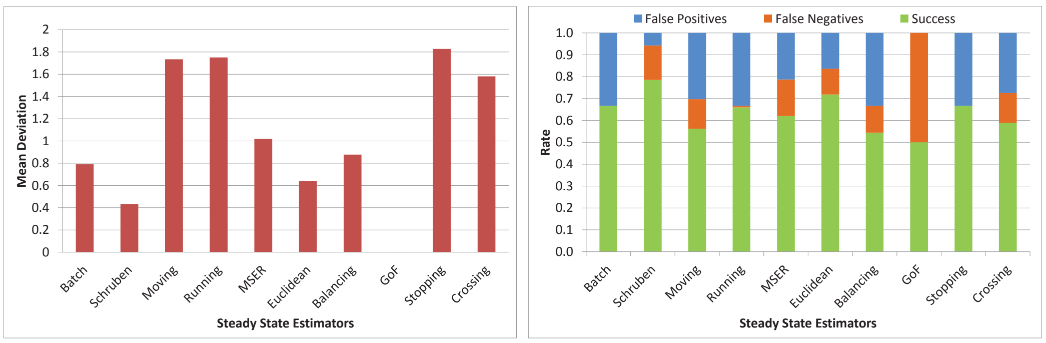


Figure 6.2.: Mean deviation and success rate of the tested steady state estimators applied on problem data.

behave differently. The goodness of fit steady state estimator has by far the best deviation of less than 0.002, but also the worst success rate with 0.5. The good accuracy of this estimator might result from the rare cases (which might be more easy to handle) where it finds the warm-up phase's end. This explanation is supported by the fact that the cases where this estimator is unsuccessful are all false negatives.

The stop crossing and batch means estimators seem to have an opposite behavior. Both produce false positives in cases where they are unsuccessful in finding the end of the warm-up phase. However, the success rate of more than 0.5 indicates, that they do not always find an end of the warm-up phase and that they are very reliable in these cases. This could be an important feature for a selection & composition algorithm to consider.

Schruben's steady state estimator has the best success rate (0.78) of all tested estimators and the second best deviation (0.43) The remaining steady state estimators represent different trade-offs between success rate and deviation.

A further examination of the estimators performance on trajectories with selected features is given in the following. The features noise, bias length, and problem length are investigated. All experiment results are listed in Appendix C.

6.1.5.1. Evaluation Results with Different Noise Levels

Figure 6.3 shows the results of the steady state estimators on time series with 0 and 10 percent noise. For the MSER steady state mean estimator, the fact whether noise exists at all, has a high impact on

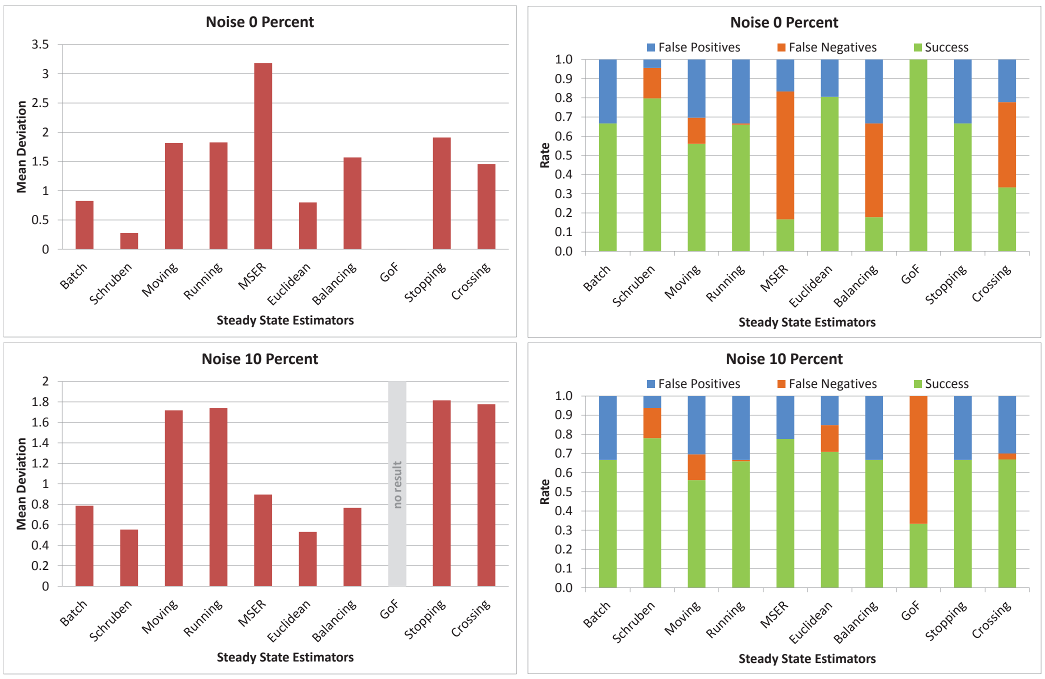


Figure 6.3.: Mean deviation and success rates of the tested steady state estimators applied on time series with different amounts of noise.

the estimation results. This is visible at the mean deviation (between estimated and real steady state) being much higher on time series without noise than with noise (independently from the concrete amount of noise). Furthermore, if noise exists, the MSER estimator has a much better success rate and does not produce false any negatives. With respect to success rates, this estimator is the most reliable of the tested estimators, when working on time series that comprise noise. Only the euclidean distance estimator is slightly better (< 3 percent) on time series with a noise of 5 percent, however it performs worse as the noise is increased. Similar to the MSER estimator, the crossing mean estimator performs better with noise than without noise, as its success rate rises from 0.33 without noise to above 0.65 if noise exists in the time series.

In contrast to MSER and crossing mean estimator, the goodness of fit estimator performs better without noise than with noise. It has a success rate of 1.0 on time series without noise and does not find any steady state in time series where noise exists (unsuccessful cases are always false negatives).

The performance of Schruben's estimator depends on the noise, as its mean deviation between estimated steady state mean and real steady state rises with the noise level in the time series it is applied to.

All in all, the amount of noise seems to be an interesting feature for finding the most suitable steady state estimator. For instance, if no noise can be found it might be reasonable to use the goodness of fit estimator to find the end of the warm-up phase. With low amounts of noise (around 5 percent) the euclidean distance estimator should be used, while MSER could be a good choice with higher amounts of noise. However, noise needs to be measured properly to make such decisions. This can be challenging, if the noise cannot be derived from the time series generator, which is the case in most real-world scenarios.

6.1.5.2. Evaluation Results with Different Problem Lengths

The results of the steady state estimators on time series with 100 and 1,000 data points are depicted in Figure 6.4. The problem length has a considerable influence on the success rates of the euclidean

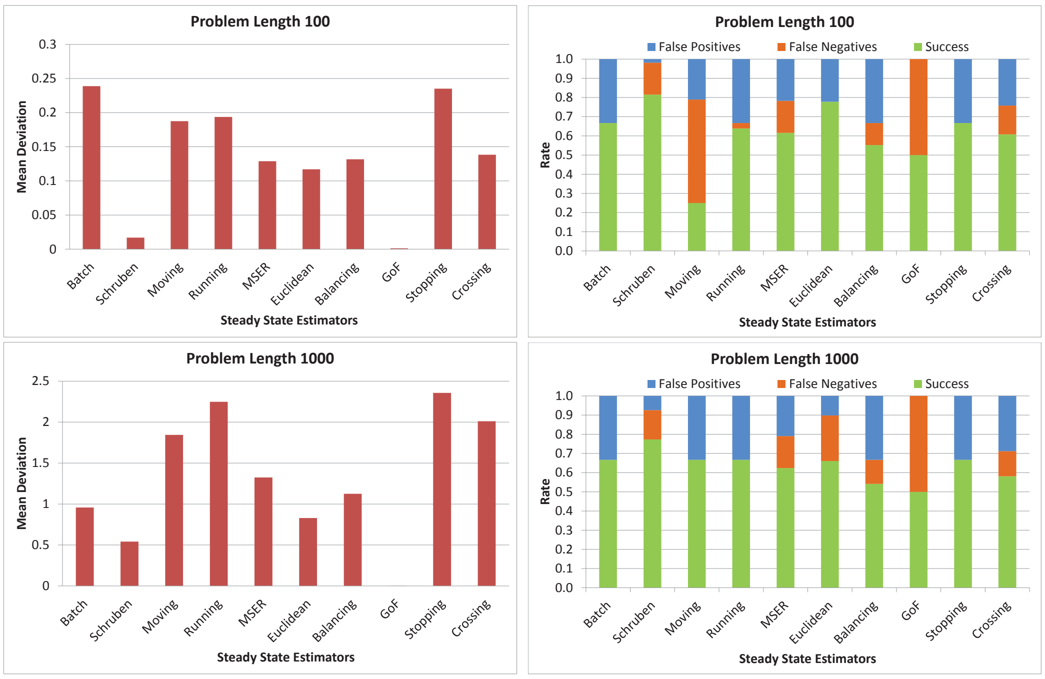


Figure 6.4.: Mean deviation and success rates of the tested steady state estimators applied on time series with different lengths.

distance and moving windows steady state estimators. For the euclidean distance estimator, the success rates gets worse, as the problem length is increased. Furthermore, it produces more false negatives with bigger problem lengths. It produces no false negatives with problem lengths of 100 and over 23 percent with problem lengths of 1,000.

The moving windows estimator has a better success rate with bigger problem lengths. With a length of 100 the success rate of 0.25 is very low, but rises to 0.67 with the other tested problem lengths.

An any case, none of both estimators has the best success rate, as Schruben's estimator performs best in all cases. Nevertheless, it might be reasonable, to use moving windows or euclidean distance estimator for a second opinion on the end of the warm-up phase, depending on the problem length of the time series under investigation

An additional interesting observation, is the rising mean deviation of all tested estimators, as problem lengths are increased. This can be explained by the fact, that the (relative) slope of the quadratic bias trend depends on the problem length. However, as all estimators behave similarly in this case, this phenomenon seems not exploitable for selection & composition.

6.1.5.3. Evaluation Results with Different Bias Lengths

Figure 6.5 shows, the evaluation results with different bias lengths. It reveals, that some of the tested estimators are unsuccessful in detecting whether the end of the warm-up phase has been reached in the

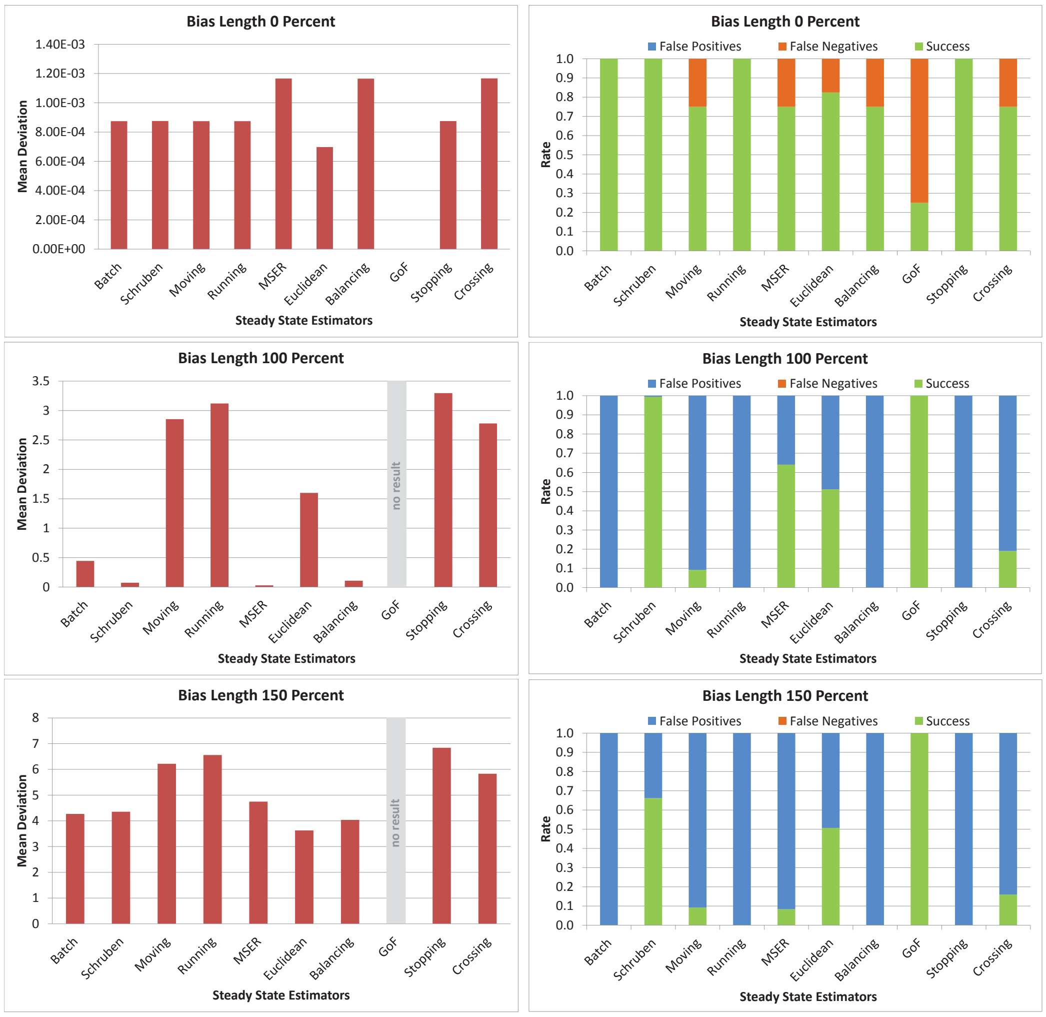


Figure 6.5.: Mean deviation and success rates of the tested steady state estimators applied on time series with different bias lengths.

generated time series, or not. For example, the batch means, running mean, and stop crossing mean steady state estimators always detect an end of the warm-up phase (producing only false positives, if it has not been reached). As identified before, the goodness of fit estimator has a tendency to reject, which makes it perfectly successful on time series with bias lengths of 100 percent and above, but also not suitable for bias lengths below. All of these estimator do not give a reliable hint, whether the end of the warm-up phase has been reached, i.e., if they require additional information to give a good estimate for the steady state mean. The balancing mean estimator even rejects some of the time series where the warm-up phase is finished, while it does not reject any time series where this is not the case.

A very interesting (but not surprising) phenomenon is the rise of mean deviations with bias lengths above 100 percent. As the data used for estimation are completely biased, the estimates differ from the real steady state. If the bias length is below 100 percent, none of the estimators produces a mean deviation worse than 0.5. Hence, the bad deviations of some estimators in the overall results are primarily caused by their insufficient ability to find out whether the warm-up phase has been finished in a time series. Consequently, a main goal for a synthetic steady state estimator is to improve this ability.

6.1.5.4. Summary

The evaluation of the steady state estimators shows two things. First, different features of time series influence the results of the steady state estimators, especially their ability to detect the end of the warm-up phase. Second, the ability of detecting the end of the warm-up phase has an impact on the quality of steady state mean estimates, and no steady state estimator is dominating. Hence, a combination of estimators could yield better results.

A synthetic steady state estimator should be designed to optimize the warm-up phase detection ability, by exploiting the the behavior of the different steady state estimators on trajectories with different features. Therefore, the steady state estimators behavior and time series features have to be monitored and evaluated by selection & composition mechanisms.

6.2. A Synthetic Steady State Estimator

This section will describe a synthetic steady state estimator which is a realization of the SPS introduced in Section 5.3.3. It is based uses the steady state estimators described in Section 6.1.2 as base-line problem solvers. The elements of the $SPS = \langle A, \mathcal{F}_P, \mathcal{F}_S, \kappa, \varsigma \rangle$ that have been realized for creating a synthetic steady state estimator, are described. The qualities of the synthetic steady state estimator are illustrated in an evaluation experiment.

6.2.1. Base-line Problem Solvers (A)

The set of base-line problem solvers comprises the steady state estimators listed in Section 6.1.2. As mentioned before, all of the estimators implement the problem solver interface (see Section 5.2.1).

6.2.2. Problem Feature Extractors ($\mathcal{F}_P$)

To get an idea about the problem time series at hand feature extractors are used for identifying three key properties that are relevant for steady state mean estimation: the amount of noise, the problem length (i.e., the size of the time series), and the type of initialization bias in the time series. Initially, the following nine features have been considered:

1. Count of values above sample mean, to measure the amount of values increasing the positive bias
2. Count of values below sample mean, to measure the amount of values increasing the negative bias
3. Count of values below sample mean subtracted from those above sample mean, to measure whether positive or negative values dominate the time series
4. Maximum positive distance to sample mean, to measure the impact of values increasing the positive bias
5. Minimum negative distance to sample mean, to measure the impact of values increasing the negative bias
6. Maximum absolute distance to sample mean, to measure the overall bias
7. Portmanteau Test [23], to get a measure for the auto-correlation and therefore an indication whether patterns in the curve are repeated
8. Variance, to measure the amount of noise
9. Time series length

Features $1-6$ characterize the type of bias (mainly whether it is positive or negative) in the time series, feature $7-8$ give an idea about the noise, feature 9 measures the problem length. The first 9 features are normalized in order to scale with different time series sizes, i.e., features $1-3$ are divided by the size of the time series and features $4-9$ by its value range.

For a better characterization of the initialization bias in the presence of noise, additional feature extractors are realized that extract features from a smoothed trend curve of the time series. All feature extractors except the one for time series length (as no difference between the length of standard and smoothed time series exists) are applied on this curve. For creating the trend curve, an exponential smoothing algorithm [73] is used.

All in all, this leads to $2 \cdot 8 + 1 = 17$ feature extractors.

6.2.3. State Feature Extractors ($\mathcal{F}_S$)

Resulting from the evaluation of the steady state estimators (see Section 6.1.5) the *detection result* of a steady state estimator (i.e., whether it found the end of the warm-up phase in a time series or not) is of particular interest. The detection result is retrieved by a state feature extractor f_d which applies the performance measurer p_d of the SSE evaluation experiment (see Equation 6.11 of Section 6.1.4).

Hence, given the output state s of an SSE, f_d realizes the following functionality:

$$f_d(s) = \begin{cases} 1, & \text{if } s.req = false \\ 0, & \text{otherwise} \end{cases} \tag{6.12}$$

6.2.4. Selection Function (ς)

As no selection of base-line solvers is considered (i.e., all base-line estimators shall be used for composition), the selection function returns all given base-line solvers, independently from the concrete features given in the feature history H_F:

$$\varsigma(H_F, A) = A \tag{6.13}$$

6.2.5. Composition Function (κ)

For composing the results a wrapper for WEKA's J48 is used. It is an implementation of the C4.5 decision tree algorithm [152]. The basic idea behind decision trees is to identify those attributes (i.e., features) of training set instances that contain most information regarding the class attribute (i.e., end of warm-up phase detected or not). This attribute selection is done recursively, to construct a tree of nodes representing decisions on attributes. To classify an instance, the tree is traversed from the root node to a leaf node, depending on the attribute values of the instance. Each leaf is labeled with one value of the class attribute (or a value range), in this case whether the end of the warm-up phase has been reached (and a steady state mean estimation is possible) or not.

6.2.5.1. Training the Composition Function

A WEKA J48 decision tree is generated by training. To generate training data, the problem generator introduced in Section 6.1.3 is used with the according parameters shown in Table 6.1. The used base-line problem solvers are applied on these training problems and features are extracted during the process.

Problem and state features are generated by the corresponding extractors described in Sections 6.2.2 and 6.2.3. Initial experiments showed, that feature 9 of Section 6.2.2 (i.e., variance) bias the learning process of the composition function, leading to overfitted results. Hence, in the final learning process this particular feature extractors has been left out.

The objective value is a boolean, denoting whether the end of the warm-up phase has been reached in the time series, i.e., whether a steady state exists. The end of the warm-up phase is calculated by the same algorithm as used in Section 6.1.4, i.e., by calculating the minimum MSE.

6.2.5.2. Learned Decision Tree

The resulting decision tree is shown in Figure C.1 of Appendix C.7. The idea behind the tree differs from the example tree of Section 5.3.3.1, pp. 85, as it decides on the end of the problem time series' warm-up phase, instead of deciding on the steady state estimator that shall be used for creating estimation results. This approach offers further opportunities for composition, as the results of different steady state estimators can be combined after the end of the warm-up phase has been found. In the present example, they have been combined in a fair portfolio (i.e., with equal weights).

A part of the decision tree, depicted in Figure 6.6, shall be discussed, exemplarily for the whole tree, in the following.

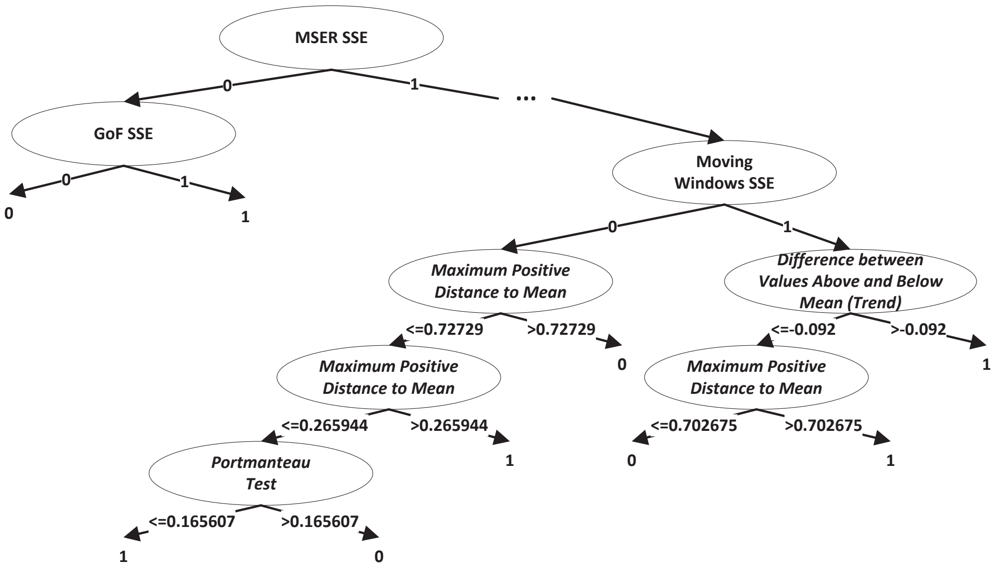


Figure 6.6.: Part of the decision tree used by the synthetic steady state estimator. A decision of 0 denotes that the end of warm-up phase has not been reached in the time series under investigation, a result of 1 denotes that it has been reached.

The root of the decision tree, i.e., the first decision to be made, is based on the MSER estimator's result — a state feature. If it did not detect an end of the warm-up phase, the final decision is left to the goodness of fit estimator. Otherwise, i.e., if the MSER estimator detected the end of an warm-up phase, a set of additional decisions has to be made, based on problem and state features, which eventually might lead to a decision based on the moving windows estimator's result. However, in any case (it detected an end of the warm-up phase or not), this is not the final decision. Additional problem features, like maximum distance to the mean, or auto-correlation (Portmanteau Test), are considered. The example shows, that different intervals have to be distinguished for each feature, e.g., the maximum positive distance to mean is first checked whether it is lower or higher than 0.72729, and if it is lower it is checked for 0.265944.

Altogether, the example shows, that problem *and* state feature are of importance for the steady state estimation, in particular detecting the end of the warm-up phase.

6.3. Comparison of Base-line Steady State Estimators and Synthetic Variant

To evaluate the synthetic steady state estimator its performance is compared to those of the base-line problem solvers, i.e., steady state estimators. Therefore, the synthetic estimator is applied to the training data time series, to show that it indeed is able to produce better results than the base-line estimators. In addition, all estimators are applied on real simulation output data, to show the benefits of the synthetic estimator with unknown data.

6.3.1. Performance on Training Data

Figure 6.7 shows the overall results on the training data, including those of the synthetic steady state estimator. Regarding the success rate, it has the best performance of all estimators with 0.88.

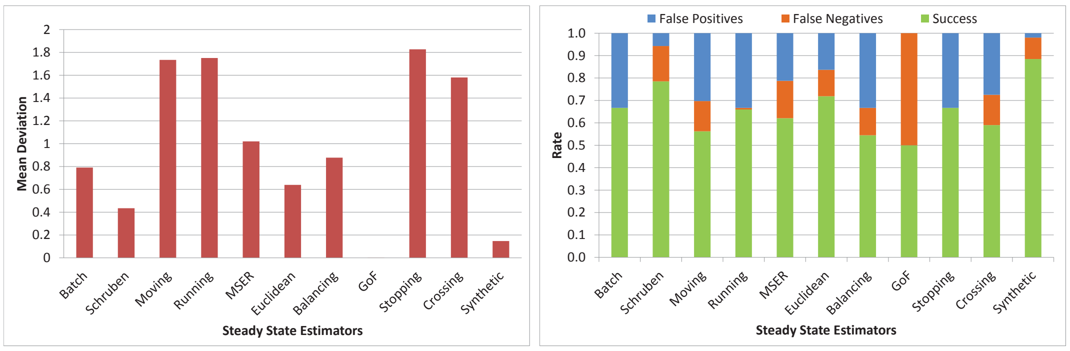


Figure 6.7.: Mean deviation and success rate of the tested steady state estimators applied on problem data.

With respect to the mean deviation between estimated steady state mean and real steady state, the goodness of fit estimator generates better results (less than 0.002 compared to a rate of around 0.15 produced by the synthetic estimator). However, keeping in mind the bad success rate of the goodness of fit estimator (0.5), this is not comparable. The goodness of fit estimator detects an end of the warm-up phase in only rare cases. Hence, the synthetic estimator has to estimate the steady state mean in many more (potentially harder) cases. It still produces a better mean deviation then the second best estimator — the Schruben's test estimator, which has a comparable success rate (around 0.8).

Altogether, the synthetic estimator produces not only the best success of all tested steady state estimator but also the best mean deviation of those estimators that show a decent capability of finding the end of the warm-up phase. However, these good results have been achieved on data used for training the composition function. A more reliable evaluation has to be be performed, by using new problem data generated in real simulation runs.

6.3.2. Performance on Simulation Output

To compare the performance of the synthetic estimator to those of the used base-line problem solvers, a set of π-Calculus models [136] has been executed. These models produce time series used as input for the steady state estimators.

Six models are used for this purpose, all of them are implementations taken from the examples of SPiM [149]:

- MgCl2: representing the behavior of a $MgCl_2$ solution. The time series contains the counts of the Mg particles over time.

- NaCl: representing the behavior of a $NaCl$ solution. The time series contains the counts of the Na^+ particles over time.
- KNa2Cl: representing the behavior of a KNa_2Cl solution. The time series contains the counts of the Na^+ particles over time.
- HCl: representing the behavior of a HCl solution. The time series contains the counts of H particles over time.
- rNHCOR: representing the synthesis reaction of an rNHCOR amide. The time series contains the counts of the $rNHCOR$ particles over time.
- MAPK: representing a simplified MAPK cascade. The time series contains the counts of the KBP cells over time.

A specific challenge for steady state estimation on simulation data, is the execution in multiple iterations. Therefore, each generated time series is processed in maximal 15 iterations, i.e., the time series are divided into 15 chunks. The first 5 chunks comprise the first time points of the time series, until the *ideal truncation point*[2] is reached, i.e., the warm-up phase is equally divided into the first 5 chunks. The following 10 chunks comprise the next 10 segments of the time series (after the truncation point).

This kind of input data allows us to investigate the accuracy and required data of steady state estimators. Accuracy is again measured by calculating the deviation between the real steady state mean and the mean estimated by the steady state estimator. However, the success rate is not sufficient as performance criterion in this evaluation experiment, as the mere fact whether the warm-up phase has ended is of minor interest. It is more important to know, how much data a steady state estimator needs for producing reliable results. This amount of required data is measured by counting the required time series chunks until the estimator produces a result.

For creating reliable results, each estimator application has been replicated 10 times. Hence, the results presented in the following are mean values.

6.3.2.1. Results

The accuracy of problem solvers is depicted in Figure 6.8. From the base-line problem solvers, only the results of Moving Window, Batch Means, and Schruben's Test steady state estimators are presented, as they performed best among the 10 tested base-line steady state estimators during the evaluation experiment. The results of the remaining estimators is depicted in Table S1 of the supplementary material. The results of the rNHCOR model are not discussed, as none of the executed estimators was able to estimate a steady state on 15 time series chunks generated by this model.

The SPS instance is the most accurate estimator on all time series but those generated by the HCl model (where it is the second best). On time series generated by this model, all estimators produce very accurate results with less than 1 percent deviation from the real steady state mean. Hence, the worse performance of the SPS instance could be explained by statistical fluctuations resulting from the closeness of the different estimates.

Figure 6.9 shows the required chunks of data points until a steady state could be estimated. Negative values correspond to the first 5 chunks of the time series (-5 being the first chunk, -4 the second etc.), i.e., the corresponding problem solver gave a steady state estimate on time series data that did not finish their warm-up phase. Hence, negative values indicate that the problem solver is less robust, as it returns biased results. On the other hand, positive values correspond to the next 10 chunks of the time series (1 being the first chunk after the truncation point, 2 the second etc.), i.e., high positive values indicate that the estimator requires many data points after the ideal truncation points to give an estimate, making it less efficient.

[2]The ideal truncation point is again calculated by identifying the minimal MSE

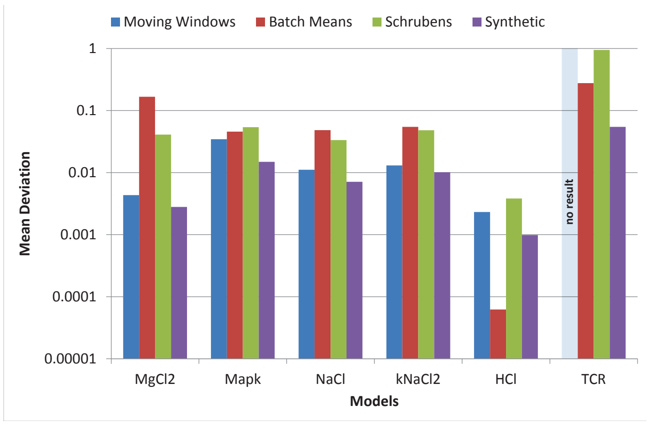


Figure 6.8.: The accuracy of the steady state estimators. It is measured as relative distance between real and estimated steady state on the time series produced by the evaluation models. Note that the deviation axis is scaled logarithmically.

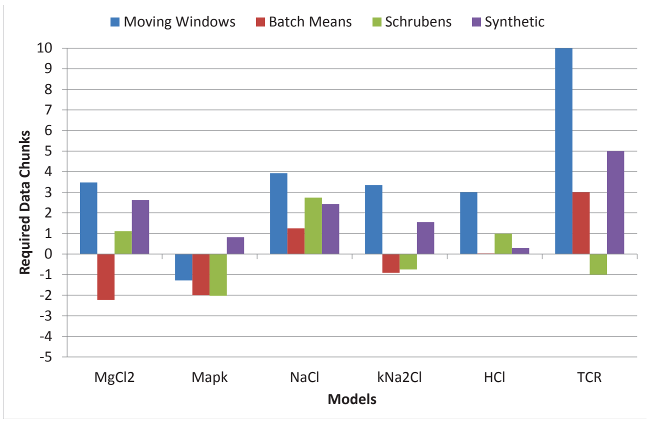


Figure 6.9.: The number of required iterations the steady state estimators needed. for producing an estimate on the time series produced by the evaluation models. Negative values correspond to chunks of data points before the ideal truncation point. Positive values correspond to chunks of data points after the ideal truncation point.

The SPS instance is the only estimator where the number of required chunks is higher than 0 for all models, which shows its robustness. While estimators exist that require less chunks after the ideal truncation point, the SPS instance is at least the second best estimator in this regard and, hence, moderately efficient.

Altogether, the generated SPS instance is the most accurate steady state estimator in the evaluation experiment. It outperforms the other estimators when applied to input data created by all but one model. In addition, it is the most reliable estimator and its efficiency (i.e., required data chunks) is acceptable. While the run time of the SPS instance might be higher than that of a base-line steady state estimator, the impact on overall experiment duration is usually negligible. This is because the generation of time series by simulation typically requires much more computing time than their analysis.

6.4. Summary

This chapter presented a case study dealing with the generation of an *SPS* instance for steady state estimation. Therefore, existing steady state estimators have been analyzed in a performance experiment. Problem data (i.e., time series) have been created by a problem generator covering characteristics relevant for steady state estimation. Analysis of the data showed, that the estimators perform differently depending on these characteristics, implying that a composition of estimators might be beneficial.

Consequently, an *SPS* instance has been trained which uses the existing steady state estimators as base-line problem solvers, and feature extractors for getting information about problem time series and base-line problem solver results. The composition of estimators has been done by a decision tree, which considers these features to denote whether the end of the warm-up phase has been reached in the time series under investigation. This decision tree is trained by using the data of the performance experiment.

When applied on the training data, the synthetic estimator outperformed all used base-line estimators. On previously unseen simulation output data, it did not show this superior performance but still had benefits regarding the amount of required data.

While these results are promising, the construction of well-performing estimator ensembles is non-trivial and there are still many potential improvements, e.g., regarding their accuracy. Although the *SPS* instance reliably found the end of the warm-up phase on the tested simulation output data, it is still possible that it fails on data that is outside the considered problem region. The results should be re-checked with alternative simulation outputs. If this deteriorates performance, the ensemble approach could be extended towards regression, i.e., other machine learning methods that directly estimate a steady state mean by considering time series features. Moreover, additional features, i.e., time series properties and results of other steady state estimators, may further improve the performance of such ensembles. In a similar vein, the application of other machine learning methods could be of interest. Also note, that the results are empirical (in the spirit of [91]); a more formal treatment of the approach may allow to gain further insights (e.g., regarding the role of noise).

7. Case Study II: Synthesizing a Statistical Analyzer for Replication Number Estimation

"It's the repetition of affirmations that leads to belief. And once that belief becomes a deep conviction, things begin to happen."

Muhammad Ali

The statistical analysis of simulation output data is usually required to tackle stochasticity in the investigated model. Output data from multiple replications have to be aggregated, e.g., by calculating the mean. To gain trustworthy results, the statistical analysis method has to give feedback on how many replications have to be executed. The number of required replications can be estimated, e.g., by considering confidence intervals.

This chapter deals with methods that aim at calculating the mean of replication results within a given confidence interval, and have been used for multi-run analysis in the example experiments of Section 3.3. Methods are applicable, which differ basically in their strategy, i.e., iterative or two-stepped [7, p. 71-73], and the used distribution, i.e., normal [25] or student-t distribution [168] for estimating the number of required replications.

The study, described in the following, shall show that the algorithm selection & composition concept introduced in Chapter 5 is not restricted to single-run analysis methods like steady state estimation, but can be applied at different levels of the simulation experiment — multi-run analysis in this case. As the *principle* applicability of the SPS shall be shown, the study will not be as extensive as the one for steady state estimation discussed in Chapter 6.

The chapter begins with the description of an SPS that is, afterward, compared to its base-line statistical analyzers when applied on evaluation problems. Two alternative composition functions are presented, each generated with respect to one of two different aims: reducing the amount of required replications (i.e., the sample size), and minimizing the number of analysis iterations. The second aim is interesting to enable an optimal parallelization of replications in a simulation experiment, as only replications that are scheduled in the same iteration can be executed in parallel. As a result of the two aims (and composition functions), two SPS instances are created.

7.1. Synthetic Statistical Analyzers

This section presents two synthetic statistical analyzers which are instances of the SPS introduced in Chapter 5.3.3. The SPS instances differ in their composition functions, which are tuned for minimizing either the amount of required replications, or the number of analysis iterations. The elements of the $SPS = \langle A, \mathcal{F}_P, \mathcal{F}_S, \varsigma, \kappa \rangle$ are described, in the following.

7.1.1. Base-line Problem Solvers (A)

The base-line statistical analyzers calculate the mean of simulation (analysis) outputs and differ primarily in their used strategy, i.e., whether they work two-stepped or iterative [7, p. 71-73].

A two-stepped procedure requests an initial sample X_0 with sample-size n_0 and estimates the amount of required replications, i.e., the required sample-size n_{i+1} according to:

$$n_{i+1} = \frac{Dist_c^2 \cdot \sigma_{X_i}^2}{e^2}, \tag{7.1}$$

where $dist_c$ is the c-quantil of distribution $dist$, σ_{X_i} is the standard deviation of sample X_i (i.e., the sample that has been generated as answer for the previous analysis iteration), and e is the allowed error tolerance. Note that this procedure gives an *estimate* for the amount of required replications. Hence, the number of replications might be initially underestimated, and the estimation process has to be repeated multiple times. Consequently, the implementation of this procedure uses more than two steps and terminates as soon as the number of estimated replications n_i is less or equal to the number of executed replications.

The iterative procedure requests a fixed (usually small) sample-size n_i in multiple iterations and monitors the confidence interval according to:

$$Dist_{1-\alpha/2} \cdot \sigma_X \leq e \cdot \sqrt{N}, \tag{7.2}$$

where N is the amount of all executed replications, so far, i.e., $N = \sum_{i=0}^{k} n_i$, with k being the number of analysis iterations. The procedure is finished as soon as the above condition is met.

Both procedures, two-stepped and iterative, ensure that the calculated mean of replication results falls into a given confidence interval. Therefore, they rely on the c-quantil of a distribution $dist$, as well as a and an error rate e. In this case study, two alternatives for $dist$ — the normal and student-t distribution — are used, whereas c is set to 0.95 and e is set to 0.05, constantly.

Altogether, four (parametrized) base-line statistical analyzers are used for composition. They result from the combination of the two strategies and the two distributions. The steps in the iterative procedures have been set to 100, i.e., in each iteration where the procedures did not achieve the desired confidence interval, they request 100 additional replications.

7.1.2. Problem Feature Extractors ($\mathcal{F}_P$)

12 features have been considered to identify the characteristics of samples and their underlying distributions:

1. Mean
2. Median
3. Value Range
4. Count of values above sample mean
5. Count of values below sample mean
6. Count of values below sample mean subtracted from those above sample mean
7. Maximum positive distance to sample mean
8. Minimum negative distance to sample mean
9. Maximum absolute distance to sample mean
10. Skewness
11. Anderson-Darling Test for normality
12. Variance, to measure the amount of noise

Features $1-3$ are extracted to get an idea about the values of the sample (and the underlying distribution), whereas features $4-10$ give hints about its shape. Feature 11 tests whether the sample originates from a normal distribution, whereas feature 12 (the variance) is used to measure the amount of noise in the sample.

7.1.3. State Feature Extractors ($\mathcal{F}_S$)

Synthetic statistical analyzers are generated regarding two aims, i.e., to minimize the number of required replications, and to minimize analysis iterations. Both aims depend on the required replication results that are requested at the end of an analyzer iteration.

Hence, a state feature extractor f_d is required that returns this request, i.e., the replication number estimate. Given the output state s of a statistical analyzer, f_d realizes the following functionality:

$$f_d(s) = s.req \tag{7.3}$$

Remember that the dot-notation refers to sub-elements of a tuple, in this case the request req of state s.

7.1.4. Selection Function (ς)

As no selection of base-line solvers is considered (i.e., all base-line estimators shall be used for composition), the selection function returns all given base-line solvers, independently from the concrete features given in the feature history H_F:

$$\varsigma(H_F, A) = A, \tag{7.4}$$

7.1.5. Composition Function (κ)

Similar to the composition function of the synthetic steady state estimator described in Section 6.2.5.1, a Weka J48 decision tree is used for composing the statistical analyzers. For training the decision tree, training samples are generated by a problem generator that returns random numbers according to a given probabilistic distribution.

The used distributions and their parameters are depicted in Table 7.1. The base-line statistical analyzers are applied on these training samples, and features are extracted to be evaluated in the following learning process. Two composition functions are learned.

7.1.5.1. Learned Decision Trees

The composition functions for statistical analysis, described in the following, work in one stage by applying decision trees that directly return a statistical analyzer, whose result and request are directly used as result and request of the SPS instance. This differs from the composition function for steady state estimation, described in Section 6.2.5.1, pp. 107, which works in two stages, i.e., it applies first a decision tree for deciding on the end of the warm-up phase, and second a portfolio for composing the steady state mean estimate if the end has been detected.

Figure 7.1 shows the decision tree that has been trained to minimize the amount of replications. The result statistical analyzers that shall be used by the composition function are stored in the leaves of the decision tree, whereas the other nodes denote the features that are checked for deciding which analyzer to use. All leaves in the tree comprise an iterative procedure. This is not surprising, as they request a small number of 100 replications in each iteration, whereas the two-stepped approaches usually request more replications in their second step (depending on the desired confidence interval and error tolerance). Hence, while the two-stepped approaches give sophisticated estimates about the number of required replications, they also might overestimate it too much, whereas the iterative procedures approximate the number of required replications more cautiously. As a consequence, the

Distribution	Parameters	
Uniform		
Normal	Expectation μ	standard deviation σ
	0.0, 0.1, 0.2, 0.3, 0.4, 0.5 0.6, 0.7, 0.8, 0.9, 1.0	0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
Exponential	Rate parameter λ	
	1.0 , 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0	
Gamma	Rate parameter λ	
	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	
Poisson	Rate parameter λ	
	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	
Weibull	Shape parameter k	Scale parameter λ
	1.0 , 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0	1.0 , 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0
Erlang	Shape parameter k	Scale parameter λ
	1.0 , 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0	1, 2, 3, 4, 5 6, 7, 8, 9, 10
LaPlace	Shape parameter k	Scale parameter λ
	1.0 , 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0	1, 2, 3, 4, 5 6, 7, 8, 9, 10
Poisson	Rate parameter λ	
	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	
Gamma	Humps	Hump size
	2, 3, 4, 5, 6, 7, 8, 9, 10	0.25, 0.3, 0.35, 0.4, 0.45, 0.5 0.55, 0.6, 0.65, 0.7, 0.75
	Hump domain	
	0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0	

Table 7.1.: Distributions and parameters for generating the problem samples.

decision tree prefers analyzers that are based on iterative procedures as leaves (which denote the analyzer, whose results shall be used by the SPS instance).

Nevertheless, the tree considers the state feature (i.e., the request, see Section 7.1.3) of a two-stepped analyzer, which shows that, even though this analyzer does not produce optimal results, it can still be relevant for getting valuable information about the underlying problem.

Figure 7.2 shows the decision tree that has been trained to minimize the analysis iterations. In contrast to the tree depicted in Figure 7.1, this tree comprises only two-stepped analyzers in its leaves. This is not surprising as well, since such analyzers usually request a large number of replications (which is mostly close to the overall number of required replications) in their second analysis iteration. This leads to many requested replications in few analysis iterations — a behavior that has been recognized by the learning procedure.

Altogether, the decision trees show that the learning procedure recognizes which analysis algorithms are best suited for which aim, i.e., iterative analyzers for minimizing the number of required replications and two-stepped analyzers for minimizing the amount for analysis iterations. While these findings are rather obvious, they are sufficient to illustrate the *principle* applicability of the SPS on multi-run analysis algorithms, which is the main reason for conducting the present study. More complex studies, e.g., involving the generation of a decision tree that has been tuned towards a combination of both aims, could be interesting for future work.

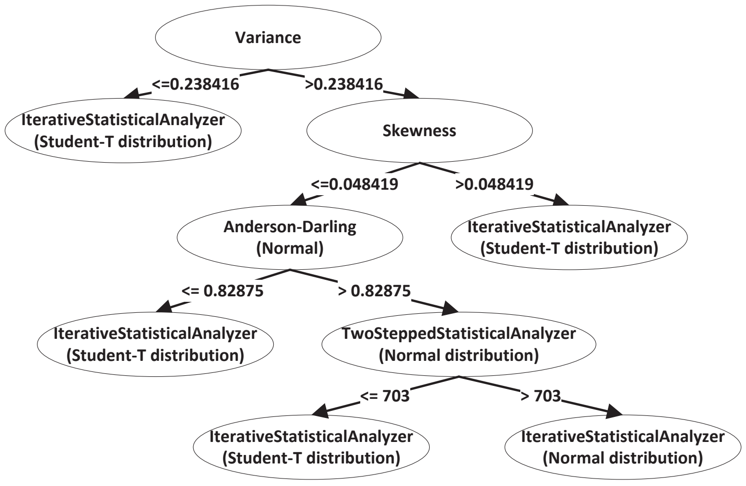

Variance
<=0.238416
>0.238416
IterativeStatisticalAnalyzer (Student-T distribution)
Skewness
<=0.048419
>0.048419
Anderson-Darling (Normal)
IterativeStatisticalAnalyzer (Student-T distribution)
<= 0.82875
> 0.82875
IterativeStatisticalAnalyzer (Student-T distribution)
TwoSteppedStatisticalAnalyzer (Normal distribution)
<= 703
> 703
IterativeStatisticalAnalyzer (Student-T distribution)
IterativeStatisticalAnalyzer (Normal distribution)

Figure 7.1.: The generated decision tree, for minimizing the number of estimated replications.

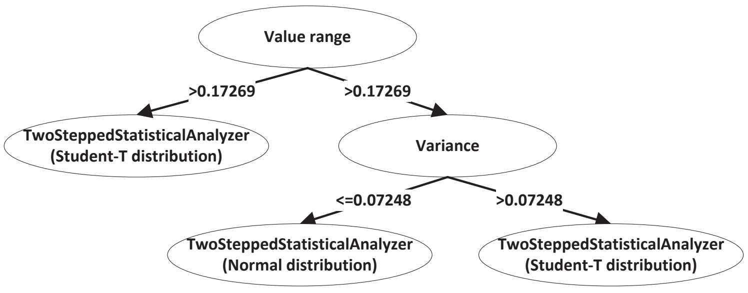

Value range
>0.17269
>0.17269
TwoSteppedStatisticalAnalyzer (Student-T distribution)
Variance
<=0.07248
>0.07248
TwoSteppedStatisticalAnalyzer (Normal distribution)
TwoSteppedStatisticalAnalyzer (Student-T distribution)

Figure 7.2.: The generated decision tree, for minimizing the number of analysis.

7.2. Comparison of Base-line Statistical Analyzers and Synthetic Variants

Corresponding to the aims of the generated synthetic problem solvers, two performance criteria are considered. The overall amount of required sample sizes, as well as the required analysis iterations have been measured.

Estimated sample sizes N_j (with j being the index of an analyzer application) have been normalized by calculating the mean relative deviation $\overline{d}$ to the ideal sample sizes, i.e., the minimum number min_j of required data points to gain a confidence interval of 0.95 and an error tolerance of 0.05:

$$\overline{d} = \frac{1}{m} \cdot \sum_{j=1}^{m} \frac{|N_j - min_j|}{min_j}, \tag{7.5}$$

where m is the overall number of analyzer applications.

Required iterations have been normalized by calculating the mean ratio between iteration count $it(j)$ and estimated sample size N_j, as the sample size represents the maximum possible amount of iterations (in case that each iteration requests for exactly one additional replication):

$$\overline{r} = \frac{1}{m} \cdot \sum_{j=1}^{m} \frac{it_j}{N_j} \tag{7.6}$$

Both performance criteria have been measured during the evaluation of the statistical analyzers on the training data, as well as on simulation output. The results are depicted in the tables of Appendix D and will be discussed in the following.

7.2.1. Performance on Training Data

Figure 7.3 shows the overall results on the training data described in Section 7.1.5.

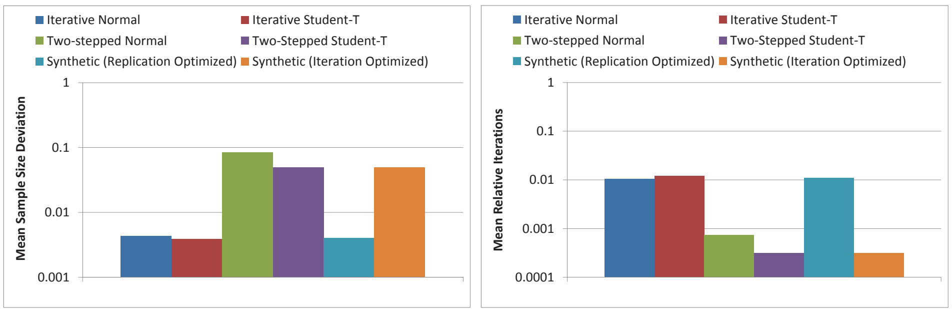


Figure 7.3.: Mean deviation to ideal sample sizes (left), and relative iterations (right) of the different statistical analyzers, applied on the training data. Sample sizes have been normalized by calculating the mean deviation to the ideal sample sizes, i.e., the minimum number of required data points to gain a confidence interval of 0.95 and an error tolerance of 0.05. Iterations counts have been normalized by dividing them through the sample sizes.

The results show that the synthetic statistical analyzers are well trained for their purpose. The *SPS* instance trained for minimizing the amount of replications requires similar sample sizes as the iterative statistical analyzers and showed the best behavior in this regard. The *SPS* instance trained for minimizing the amount of iterations, on the other hand, requires similar iteration numbers as the two-stepped procedures, and belongs to the best analyzers for this purpose.

7.2.2. Performance on Simulation Output

For evaluating the statistical analyzers on simulation data, the same models as in Section 6.3.2 have been used to generate problem data, by observing the described species. For each model, the final state after $10,000$ simulation steps (where the warm-up phase is finished in all models) has been recorded and the count of the species used as sample point for the statistical analyzers.

Figure 7.4 depicts the mean required sample size each analyzer took on simulation output produced by the different models. The results confirm the observations made on the training data results. The iterative analyzers are the best base-line analyzers for minimizing the amount of data points, and the synthetic analyzer trained for this purpose exploits this.

Similar, the two-stepped analyzers outperform the other base-line analyzers with respect to required analysis iterations, as depicted in Figure 7.5. Thereby, the two-stepped analyzer relying on the student-t distribution has the best performance. The synthetic analyzer trained for this purpose selects this analyzers and relies on its results.

Altogether, the evaluation results show that the learning process could identify dominating base-line analyzers that are exploited by the resulting *SPS* instances.

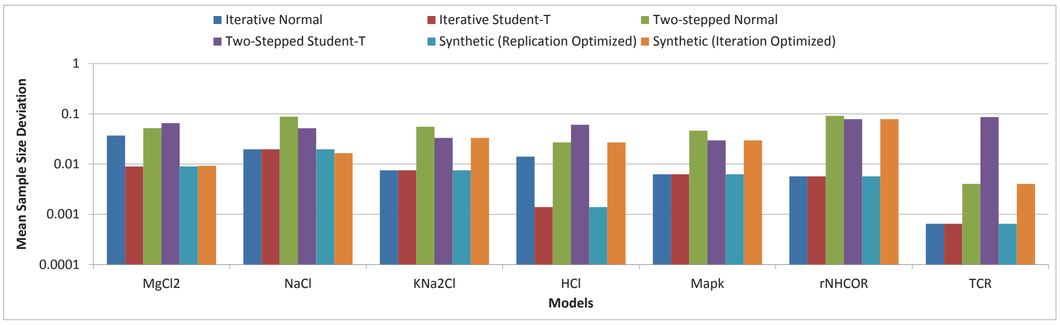


Figure 7.4.: Mean deviation to ideal sample sizes of the different statistical analyzers, applied on the simulation output of different models. Sample sizes have been normalized by calculating the mean deviation to the ideal sample sizes, i.e., the minimum number of required data points to gain a confidence interval of 0.95 and an error tolerance of 0.05.

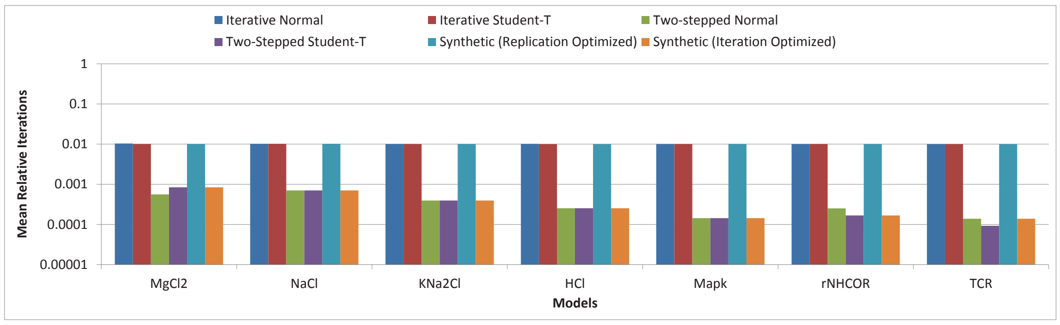


Figure 7.5.: Required iterations of the different statistical analyzers, applied on the simulation output of different models. Required iterations have been normalized by dividing them through the sample sizes.

7.3. Summary

This chapter presented a case study for generating and evaluating *SPS* instances for statistical analysis of simulation results. The *SPS* instances were based on four statistical analyzers as base-line problem solvers that used two different procedures, two-stepped and iterative, and two different probabilistic distributions, normal and student-t, for estimating the amount of required replications. The used procedure turned out to have high influence on the measured performance, i.e., the overall amount of required replications and the number of required analysis iterations. Iterative procedures had advantages when minimizing replications, whereas two-stepped procedures were beneficial for minimizing the number of required iterations. The learning process used different probabilistic distributions for generating training data, recognized this behavior, and exploited it to build *SPS* instances for the two different objectives. Similar to the synthetic steady state estimator of Chapter 6, decision trees have been generated for realizing composition functions.

The study showed that the SASF extension is applicable on the multi-run level, in addition to the single-run level as shown with the synthetic steady state estimator in Chapter 6. However, as the study focused on the principle applicability and not on a detailed examination of *SPS* features for this experiment level, improvements are possible. Such improvements could include the consideration of different learning techniques that combine both pursued aims, i.e., minimizing replications and analysis iterations, in one composition function. Furthermore, additional aims or performance criteria might be considered, e.g., execution time or distance between estimated mean and expectation value of the probabilistic distributions that underlie the simulation replication results of the evaluation models.

Especially the latter is challenging, as the underlying distributions of simulation run replications are not easily identified.

Part III.

Conclusion and Appendices

8. Concluding Remarks

"I seldom end up where I wanted to go, but almost always end up where I need to be."

Douglas Adams

Executing a simulation experiment is a challenging task. Awareness about the different methods, that are exploited in the field, is important for deciding when to apply which method.

This thesis proposed a concept for guiding users in conducting simulation experiments, by reducing the amounts of decisions to be made. Two approaches have been followed, therefore. The first approach structures the experiment to assist users in deciding which principal tasks have to be considered, while the second approach assists users in finding the right algorithms to execute those tasks. Approaches realizing such ideas have been implemented in the GUISE environment. This chapter will summarize the approaches and give an outlook about future work.

8.1. Summary

The first part of this thesis deals with reducing the amount of decisions in simulation experiments by structuring them. Therefore, Chapter 2 reviews literature and tools on the topic to identify principal tasks. As a result, six tasks are derived:

- specification — defining the experiment
- configuration — selecting interesting model parameters
- model execution — executing the model
- data collection — collecting data of interest
- analysis — analyzing the collected data
- evaluation — assessing the analysis results

Due to distinguishing these tasks, the experiment gets a clear and explicit structure which makes it more transparent to the user and supports guidance. The experiment structure, as well as the underlying tasks, comply with different types of simulation experiments, e.g., validation or optimization experiments. The tasks are general, i.e., additional more concrete (sub-)tasks might be required in specific experiments, e.g., depending on the executed model, the simulation task might comprise different parts like random number generation, event queue handling, etc.

To make the experiment structure accessible, it is implemented as an extension of the experimentation layer that is part of the plugin based M&S framework JAMES II designed by Himmelspach et. al. [85]. Tasks are realized as JAMES II plugin types to maximize flexibility and extensibility, while concrete methods are integrated as plugins and through clear interfaces. The extension adds plugin types for the tasks configuration, analysis and evaluation. It builds on the basic functionality for executing simulation studies, as well as plugin types for the tasks model execution and data collection that ship with the standard experimentation layer of JAMES II. JAMES II's parameter blocks are exploited to create experiment specifications in a flexible manner that forms a basis for alternative specification methods, e.g., a GUI or specification language, which can be translated to parameter blocks. Four example experiments (see Section 3.3), differing in their type (validation, sensitivity analysis, optimization, and exploratory analysis) as well as application domain (molecular biology, cell biology, and electrical engineering), illustrate the extended experimentation layer's flexibility and applicability.

As argued by Rybacki et. al. [160], a workflow implementation of the M&S processes contributes to the credibility of simulation studies, by providing user guidance and documentation features, and might help to overcome the "crisis of credibility" of simulation studies [146, 118]. Hence, a workflow mapping of the experiment concepts is described in Chapter 4. The mapping exploits the notion of templates and frames of the workflow management system WorMS [160]. Corresponding to the plugin-based design of JAMES II, templates are mapped to plugin types, i.e., tasks, and frames to plugins, i.e., methods like simulation algorithms or steady state estimators. Templates and frames help to fully exploit the potential and flexibility of the experiment structure and to make it accessible to users. The applicability of the approach is shown by realizing the example experiments of Section 3.3 as workflow (see Section 4.3).

To answer the second question, i.e., deciding which algorithms shall be used to execute a task, algorithm selection & composition mechanisms are integrated into the experiment structure. The overall composition process is automated, but triggered manually by the user, which is necessary, as the user needs to define, e.g., the performance metric of interest, or the problems to be used for training the selection & composition mechanism. While the need for manual intervention could be perceived as a drawback, it also helps a practitioner to be aware of the available algorithmic alternatives and the underlying assumptions of their composition, e.g., that the training set of problems is representative for the tasks to come.

To integrate selection & composition mechanisms into the experimentation layer, the *Simulation Algorithm Selection Framework (SASF)*, developed by Ewald [48], is extended, in Chapter 5, to compose simulation experiment methods that are fine-tuned toward their application domain. The extension is based on the notion of a synthetic problem solver (SPS) component, which incorporates *base-line* problem solvers (i.e., algorithms), e.g., steady state estimation methods, and combines them to improve the overall performance. It works on the assumption that each base-line problem solver can solve problems of the given type alone, and that all solvers for this problem type work on the same input data, e.g., time series in case of steady state estimation. To unify the handling of different algorithm types, a general problem solver interface is introduced, with which various algorithms used in simulation experiments comply. Different problem solver types used in a simulation experiment are mapped to that interface, illustrating its generality.

The selection & composition approach involves two major steps: a) it evaluates the base-line problem solvers on a set of representative problems, and b) it analyzes the collected data to generate an SPS instance with superior performance, e.g., in terms of robustness. To prototype the creation and usage of synthetic problem solvers, the JAMES II plugin system is adapted to accommodate them as synthetic plugins. By extending the plugin system, a high degree of flexibility with respect to both the base-line algorithms and the mechanisms for combining them, is ensured. To show the generality of the approach, existing algorithm selection & composition techniques are discussed with respect to realizing them as synthetic problem solvers. The applicability of the synthetic problem solver is shown in two case studies concerning different problem solver types.

The first study (see Chapter 6) deals with making simulation-based steady state estimation [7, p. 96] more accessible to users who lack the experience in selecting a suitable method, by joining several steady state mean estimators to a more robust estimator with superior performance. After giving a formal definition for steady state mean estimators as used in this work, ten well-known estimators are applied to a set of synthetic problem trajectories and their performance is analyzed. As the approach relies on a broad and representative set of sample data, a problem generator is designed to generate such trajectories. The parameters of the problem generator are selected in order to create input trajectories that cover different relevant characteristics for steady state mean estimation. The performance data are used to train the composition function of a synthetic problem solver for steady state estimation relying on algorithm ensembles [157], i.e., machine learning techniques for predicting which estimator is likely to perform well under which circumstances. The synthetic problem solver is evaluated against formerly unseen trajectories generated by seven biochemical models, including six simple reaction networks and a more complex T cell receptor signaling model, to show its benefits.

The second case study (see Chapter 7) is concerned with statistical analyzers, i.e., methods to calculate statistics from a set of simulation replications that are combined to a) minimize the number of required replications and b) to minimize the number of analysis iterations. As a result of the two aims, two synthetic statistical analyzers are trained. The training process is based on the performance data generated by applying four base-line statistical analyzers on problem samples from different statistical distributions. Both synthetic problem solvers are evaluated against samples generated by the same biochemical models as in the first study. The two case studies show that the SPS concept is applicable to the single-run and multi-run level of the experiment process.

8.2. Future Work

A major contribution of this work is the experiment structure, comprising six tasks, which has been identified in Chapter 2. The tasks are derived from a coarse-grained point of view and a more fine-grained guidance could be provided, by refining them, i.e., dividing them into different sub-tasks. The necessity for such divisions becomes clear by looking at the frame for the `StepPostProcessingTemplate` described in Section 4.2.4, pp. 60, where data are pre-processed before applying the concrete analysis method. An even finer distinction of sub-tasks might improve user guidance further. This, however, could lead to a loss of generality, as more specific tasks might be application dependent.

For instance in a cell biological experiment, the usual simulation output is a trajectory of variables counts over time, which needs to be translated into a time series that is used as input by many analysis methods, like steady state estimators. Translation and steady state estimation can be considered as two different sub-tasks. In many cases, however, biologists are not interested in plain steady state statistics, but, e.g., in repetitive patterns after the warm-up phase has ended. Hence, after the steady state estimation, a pattern recognition technique might have to be applied to identify, e.g., the life-cycle of a cell. In other cases, however, the steady state statistic itself might be of interest, or the simulation returns no trajectory but only its final state, requiring a completely different way of analysis and, hence, different sub-tasks.

In addition to the need for identifying more fine-grained tasks, they might have to be applied multiple times during the course of an experiment. An example for this is the analysis task, which is executed at two levels of the experiment (single-run and multi-run). Depending on the application, this could be necessary for other tasks as well, which raises the question whether guidance is deducible from tasks that do not happen at a fixed point during the course of an experiment.

To deal with such problems, *application-dependent* tasks could be integrated into a simulation tool by introducing user profiles (as done in SAFE [148] on a general level, with novice and power users). Each user profile would represent a different type of experimenter, who has to work on a set of tasks tailored to his specific needs. Such a system leads to three challenges. First, the identification of such user profiles would require extensive user studies and a flexible integration of resulting profiles which could be facilitated by using JAMES II's plugin system. The second challenge is the integration of adapted sub-tasks in the experiment workflow. This, however, should be easily possible, as the workflow presented in Chapter 4 is highly flexible due to its use of frames and templates. The third and probably most challenging problem is to make user profiles 'usable' — profiles have to be sufficiently accessible and described, so that each user can easily identify and apply the most suitable option.

The problem of identifying and maintaining application-dependent tasks coincides with the selection of sub-goals (see Section 2.3, pp. 22), which is a fundamental problem that is not tackled so far. At the moment, the user is left alone with the identification of sub-goals, i.e., goals for each experiment task, as it is assumed that he knows best which sub-goals he needs to pursue, e.g., a steady state estimation which is just one of many alternatives (others might be trajectory comparison or LTL-checking) for analyzing simulation output. However, the user might be confused with such alternatives or might not even be aware of them. A thorough guidance should assist users in the selection of sub-goals, and could exploit user profiles which would represent typical combinations of sub-goals for tasks, depending on user's professions and considering additional constraints.

Even if the experiment structure covers each aspect of simulation experimentation completely, it can only be the groundwork for guidance. To lead users through the experiment process, a proper user interface is required that makes the experiment structure accessible. Developing user interfaces, e.g., as GUI or experiment language, is the task of experiment usability and assistance research. A first step toward the integration of such interfaces has been taken by exploiting parameter blocks that are used to flexibly create experiment specifications. Alternatively, experiments can be specified by using workflows as done in Chapter 4. Nevertheless, user interfaces are required to make such specification formats accessible to users. They could be oriented on the identified experiment structure to generate parameter blocks or workflow instances and facilitate specifications. Furthermore, workflows should be extended to allow an interactive specification of experiments, i.e., users should decide during experiment execution how to handle each task so that they can react on intermediate results.

As workflow features are beneficial for supporting the experiment process, they might be also beneficial for the *SPS* creation process which is not guided so far, even though all required tools are provided in Section 5.3. As a consequence, method developers themselves have to set-up performance experiments, trigger performance analysis, activate learning methods for composition and selection function, and deploy the resulting *SPS*. It could be more useful, to integrate these steps into an automatically executed workflow. In this case, setting up this workflow is the only task to be done by hand, which includes defining a problem generator (for the performance experiments), relevant problem features, performance criteria, and a learning mechanism. The workflow, then, would be automatically executed and produce an *SPS* instance as output.

In many cases, several instances can be created for the same goal, e.g., by using problem generators that generate different samples but cover a similar problem space, or by using different learning algorithms (decision tree, neural nets, etc.) for generating the composition. Hence a selection between *SPS* instances might be required, putting meta-learning [184] into effect, where learning algorithms are applied on meta-data gained from previous learning experiments to improve performance. The principal structure for integrating such mechanics is already provided, as each *SPS* instance implements the problem solver interface, and a set of existing synthetic problem solvers can be combined in a new *SPS*. An automatic evaluation of newly created synthetic problem solvers and generation of a meta-*SPS* instance could be the final goal of an *SPS* creation workflow.

In addition to facilitating the *SPS* creation process, the limitations of the *SPS* concept have to be further investigated. Section 5.3 argued for its generality with respect to algorithm selection & composition mechanisms, and its practical applicability has been shown in two case studies. However, algorithms might exist, where a composition in an *SPS* instance is more challenging. Further experiments have to show how easily the approach can be adopted in other applications. A main problem in this regard is the identification of performance criteria, as not all algorithms can be assessed by simple measures as execution time or distance between algorithm result and desired value, e.g., the quality of the distances a time series comparison method produces is mostly subjective.

In addition to the limitations and ideas discussed above, guidance as ultimate goal can be criticized, as well. In guided processes or automatic assistance systems, a reaction on rare or unexpected events may be hindered, as users might overly rely on the course of the guided process. In the worst case, experts might refrain from using their experience and intuition that would help them in facing unexpected challenges which are not sufficiently tackled by the guided structure. Hence, creativity and ideas to improve processes might be hampered. An awareness for this issue is mandatory for the work with any structures that release users from decisions, and guidance as well as autonomous decisions have to be balanced carefully.

A. Implemented Experiment Methods

Configuration	
Aims	Methods
parameter scan	•parameter set scan (scanning a given set of parameter settings) •conditional parameter scan (scanning parameters until given condition is met)
parameter exploration	•random walk (random exploration of parameter space) •full-factorial experiment design [59] •Placket and Burman experiment design [150] •Franklin and Bailey experiment design [57] •latin hypercube sampling [132]
parameter search	•hill-climbing parameter search [158] •simulated annealing parameter search [113] •tabu-search [69, 70] •Hooke and Jeeves parameter search [90] •branch and bound parameter search [35] •particle swarm search [110] •NSGA-2 parameter search [38]
bifurcation	•sequential bifurcation [116]

Table A.1.: Methods implemented for the parameter configuration task.

Simulation	
Aims	Methods
NAM simulation	•NAM simulator
species-reaction simulation	•first reaction method [67] •direct reaction method [68] •optimized direct reaction method [197] •next reaction method [65] •tau-leaping method [154]
VulcaNoCs simulation	•VulcaNoCs simulator wrapper

Table A.2.: Methods implemented for the simulation task. Only those methods that are relevant for this work are listed. Many more have been implemented in the context of JAMES II and can be reused in GUISE.

Data Collection	
Aims	Methods
final state	•final state observer (returning the model's state at the end of a simulation run)
trajectory	•trajectory observer (returning the trajectory (i.e., time-value pairs) of a given species, generated during simulation)
temperature and performance of VulcaNoCs	•VulcaNoCs result observer

Table A.3.: Methods implemented for the data collection task. Only those methods that are relevant for this work are listed. Many more have been implemented in the context of JAMES II and can be reused in GUISE.

Single-Run Analysis	
Aims	Methods
statistical model checking	•LTL trajectory checker [53]
steady state estimation	•MSER steady state estimator [190] •euclidean distance steady state estimator [120] •goodness of fit steady state estimator [144] •balancing mean steady state estimator [45] •running mean steady state estimator [144] •batch mean steady state estimator [27] •crossing mean steady state estimator [193] •stop crossing mean steady state estimator •Schruben's steady state estimator [166]
trajectory comparison:	•maximum distance trajectory comparison •minimum distance trajectory comparison •average distance trajectory comparison •squared distance trajectory comparison •euclidean distance trajectory comparison [41]

Table A.4.: Methods implemented for single-run analysis.

Multi-Run Analysis	
Aims	Methods
statistical model checking	•probabilistic domain analyzer [42]
statistical replication estimation	•iterative estimation[7] •two-stepped estimation[7]

Table A.5.: Methods implemented for multi-run analysis.

Evaluation	
Aims	Methods
visualization	•data storage writer/plotter (storing and plotting analysis results) •Mosan evaluator [178]
parameter search feedback	•parameter search evaluator (returning parameter settings and their objective values) •pareto ranking evaluator (ranking parameter settings)
sensitivity analysis	•sensitivity evaluator (returning the sensitivity of parameters)

Table A.6.: Methods implemented for the evaluation task.

B. Experiment Specifications

B.1. Parameter Block for the Validation Experiment of the NAM Method

```
// Model location
URI modelURI = new URI("java://examples.stopi.mgcl2.Solution");

// Model configurator
ParameterBlock problemConfParams = new
    ParameterBlock(ParameterScanModelConfiguratorFactory.class.getName());
// add the model parameter values
Map<String, List<Serializable>> settings = new HashMap<>();
List<Serializable> values = new ArrayList<>();
values.add(10);
setting.put("a", values);
values = new ArrayList<>();
values.add(50);
setting.put("b", values);
values = new ArrayList<>();
values.add(100);
setting.put("q", values);
values = new ArrayList<>();
values.add(0.02);
values.add(0.2);
values.add(2);
values.add(20);
setting.put("D", values);
problemConfParams.addSubBl(ParameterScanModelConfiguratorFactory.PARAMETERS, settings);

// Simulator configurations
List<ParameterBlock> simConfigs = new ArrayList<>();
// Create simulator configurations
ParameterBlock simConfig = new ParameterBlock(NAMProcessorFactory.class.getName());
simConfig.addSubBl(NAMProcessorFactory.COLLISON_DETECTION,
    NAMTimeSteppedDetectionFactory.class.getName());
simConfig.addSubBl(NAMProcessorFactory.RNG, MersenneTwister.class.getName());
simConfigs.add(simConfig);
simConfig = new ParameterBlock(NAMProcessorFactory.class.getName());
simConfig.addSubBl(NAMProcessorFactory.COLLISON_DETECTION,
    NAMEventTriggeredFixedDetectionFactory.class.getName());
simConfig.addSubBl(NAMProcessorFactory.RNG, MersenneTwister.class.getName());
simConfigs.add(simConfig);

  ...

//Instrumenter
List<ParameterBlock> instConfigs = new ArrayList<>();
ParameterBlock instConfig = new
```

```
    ParameterBlock(NAMFinalStateInstrumenter.class.getName());
instConfigs.add(instConfig);

//Multi-run analysis
ParameterBlock multiRunParams = new
    ParameterBlock(StatisticalAnalyzerFactory.class.getName());
multiRunParams.addSubBl(StatisticalAnalyzerFactory.CONFIDENCE, 0.95);
multiRunParams.addSubBl(StatisticalAnalyzerFactory.ALLOWED_ERROR, 0.05);

//Evaluation
ParameterBlock evalParams = new ParameterBlock(DataStorageWriterFactory.class.getName());
evalParams.addSubBl(DataStorageWriterFactory.VARIABLE, "b_a");

specification.addSubBl(BaseValidator.PROBLEM_URI, modelURI);
specification.addSubBl(BaseValidator.CONFIGURATOR, problemConfParams);
specification.addSubBl(BaseValidator.SIMULATOR_CONFIGS, simConfigs);
specification.addSubBl(BaseValidator.INSTRUMENTER, instConfigs);
specification.addSubBl(BaseValidator.MULTI_RUN_ANALYSIS, multiRunParams);
specification.addSubBl(BaseValidator.EVALUATION, evalParams);
```

B.2. Parameter Block for the Sensitivity Analysis of a Wnt/β-catenin Pathway Model

```
// Model location
URI modelURI = new URI("java://examples.sr.WntCommunication");

// Model configurator
ParameterBlock problemConfParams = new
    ParameterBlock(ConditionalScanModelConfiguratorFactory.class.getName());

// add the model parameter values
Map<String, Number> baseValues = new HashMap<>();
values.put("betaNuc", 24900);
values.put("nAxin", 7);
values.put("nAxinP", 42);

...

problemConfParams.addSubBl(ConditionalScanModelConfiguratorFactory.PARAMETERS,
    baseValues);
problemConfParams.addSubBl(ConditionalScanModelConfiguratorFactory.THRESHOLD, 0.1);
problemConfParams.addSubBl(ConditionalScanModelConfiguratorFactory.OBJECTIVE, ‘‘S’’);

// Simulator configurations
List<ParameterBlock> simConfigs = new ArrayList<>();
// Create simulator configurations
ParameterBlock simConfig = new
    ParameterBlock(OptimizedDirectProcessorVarAFactory.class.getName());
simConfig.addSubBl(OptimizedDirectProcessorVarAFactory.RNG,
    MersenneTwister.class.getName());
simConfigs.add(simConfig);

//Instrumenter
List<ParameterBlock> instConfigs = new ArrayList<>();
ParameterBlock instConfig = new
```

```
    ParameterBlock(SrSpeciesInstrumenterFactory.class.getName());
instConfig.addSubBl(SrSpeciesInstrumenterFactory.SPECIES, ‘‘betaNuc’’);
instConfigs.add(instConfig);

//Single-run analysis
ParameterBlock singleRunParams = new
    ParameterBlock(EuclideanDistanceComparatorFactory.class.getName());
singleRunParams.addSubBl(EuclideanDistanceComparatorFactory.VARIABLE_NAME, "betaNuc");
//generate reference trajectory for comparison (from wet-lab data)
List<Double> referenceTrajectory = readTrajectoryFromWetLabData(dataLocation);
singleRunParams.addSubBl(EuclideanDistanceComparatorFactory.REFERENCE_TRAJECTORY,
    referenceTrajectory);

//Multi-run analysis
ParameterBlock multiRunParams = new
    ParameterBlock(StatisticalAnalyzerFactory.class.getName());
multiRunParams.addSubBl(StatisticalAnalyzerFactory.CONFIDENCE, 0.95);
multiRunParams.addSubBl(StatisticalAnalyzerFactory.ALLOWED_ERROR, 0.05);

//Evaluation
ParameterBlock evalParams = new
    ParameterBlock(SensitivityEvaluatorFactory.class.getName());
List<String> variables = new ArrayList<>();
variables.add("betaNuc");
variables.add("nAxin");
variables.add("nAxinP");

...

evalParams.addSubBl(SensitivityEvaluatorFactory.VARIABLES, variables);

//Assemble result parameter block
ParameterBlock specification = new ParameterBlock();
specification.addSubBl(BaseValidator.PROBLEM_URI, modelURI);
specification.addSubBl(BaseValidator.CONFIGURATOR, problemConfParams);
specification.addSubBl(BaseValidator.SIMULATOR_CONFIGS, simConfigs);
specification.addSubBl(BaseValidator.INSTRUMENTER, instConfigs);
specification.addSubBl(BaseValidator.SINGLE_RUN_ANALYSIS, singleRunParams);
specification.addSubBl(BaseValidator.MULTI_RUN_ANALYSIS, multiRunParams);
specification.addSubBl(BaseValidator.EVALUATION, evalParams);
```

B.3. Parameter Block for the Optimization Experiment of a Wnt/β-catenin Pathway Model

```
// Model location
URI modelURI = new URI("java://examples.sr.WntCommunication");

// Model configurator
ParameterBlock problemConfParams = new
    ParameterBlock(HookeJeevesModelConfiguratorFactory.class.getName());
// add the model parameter values
Map<String, Number> baseValues = new HashMap<>();
values.put("betaNuc", 24900);
values.put("nAxin", 7);
values.put("nAxinP", 42);
```

```java

...

problemConfParams.addSubBl(ConditionalScanModelConfiguratorFactory.PARAMETERS,
    baseValues);

// Simulator configurations
List<ParameterBlock> simConfigs = new ArrayList<>();
// Create simulator configurations
ParameterBlock simConfig = new
    ParameterBlock(OptimizedDirectProcessorVarAFactory.class.getName());
simConfig.addSubBl(OptimizedDirectProcessorVarAFactory.RNG,
    MersenneTwister.class.getName());
simConfigs.add(simConfig);

//Instrumenter
List<ParameterBlock> instConfigs = new ArrayList<>();
ParameterBlock instConfig = new
    ParameterBlock(SrSpeciesInstrumenterFactory.class.getName());
instConfig.addSubBl(SrSpeciesInstrumenterFactory.SPECIES, "betaNuc");
instConfigs.add(instConfig);

//Single-run analysis
ParameterBlock singleRunParams = new
    ParameterBlock(EuclideanDistanceComparatorFactory.class.getName());
singleRunParams.addSubBl(EuclideanDistanceComparatorFactory.VARIABLE_NAME, "betaNuc");
//generate reference trajectory for comparison (from wet-lab data)
List<Double> referenceTrajectory = readTrajectoryFromWetLabData(dataLocation);
singleRunParams.addSubBl(EuclideanDistanceComparatorFactory.REFERENCE_TRAJECTORY,
    referenceTrajectory);

//Multi-run analysis
ParameterBlock multiRunParams = new
    ParameterBlock(StatisticalAnalyzerFactory.class.getName());
multiRunParams.addSubBl(StatisticalAnalyzerFactory.CONFIDENCE, 0.95);
multiRunParams.addSubBl(StatisticalAnalyzerFactory.ALLOWED_ERROR, 0.05);

//Evaluation
ParameterBlock evalParams = new
    ParameterBlock(ParameterSearchEvaluatorFactory.class.getName());
evalParams.addSubBl(ParameterSearchEvaluatorFactory.MINIMIZE, true);
// set the cancelcriterion to 2000 configurations
ParameterBlock cancelCritParams = new
    ParameterBlock(ConfigurationCountCancelCriterion.class.getName());
cancelCritParams.addSubBl(ConfigurationCountCancelCriterion.COUNT, 2000);
evalParams.addSubBl(ParameterSearchEvaluatorFactory.CANCEL_CRIT, cancelCritParams);

//Assemble result parameter block
ParameterBlock specification = new ParameterBlock();
specification.addSubBl(BaseValidator.PROBLEM_URI, modelURI);
specification.addSubBl(BaseValidator.CONFIGURATOR, problemConfParams);
specification.addSubBl(BaseValidator.SIMULATOR_CONFIGS, simConfigs);
specification.addSubBl(BaseValidator.INSTRUMENTER, instConfigs);
specification.addSubBl(BaseValidator.SINGLE_RUN_ANALYSIS, singleRunParams);
specification.addSubBl(BaseValidator.MULTI_RUN_ANALYSIS, multiRunParams);
specification.addSubBl(BaseValidator.EVALUATION, evalParams);
```

B.4. Parameter Block for the Exploratory Analysis of a VulcaNoCs Model

```
// Model location
URI modelURI = new URI("java://examples.noc.NoCModel");

// Model configurator
ParameterBlock problemConfParams = new ParameterBlock(
       ParameterScanModelConfiguratorFactory.class.getName());
// add the model parameter values
Map<String, List<Serializable>> settings = new HashMap<>();
List<Serializable> values = new ArrayList<>();
values.add(2);
values.add(3);
values.add(4);
values.add(5);
values.add(6);
values.add(7);
values.add(8);
setting.put("layout", values);
values = new ArrayList<>();
values.add(1.0);
values.add(1.5);
values.add(2.0);
values.add(2.5);
values.add(3.0);
setting.put("‘Thresh"’, values);
values = new ArrayList<>();
values.add(2.0);
values.add(3.0);
setting.put("‘MinTempDelta"’, values);
values = new ArrayList<>();
values.add(1000);
values.add(5000);
values.add(10000);
setting.put("ActThresh", values);
values.add("BLOCK");
values.add("RES1");
values.add("RES2");
setting.put("RCcircuit", values);
problemConfParams.addSubBl(ParameterScanModelConfiguratorFactory.PARAMETERS,
       settings);

// Simulator configurations
List<ParameterBlock> simConfigs = new ArrayList<>();
// Create simulator configurations
ParameterBlock simConfig = new ParameterBlock(NoCSimulatorFactory.class.getName());
simConfigs.add(simConfig);

//Instrumenter
List<ParameterBlock> instConfigs = new ArrayList<>();
ParameterBlock instConfig = new ParameterBlock(NoCInstrumenterFactory.class.getName());
instConfigs.add(instConfig);

//Multi-run analysis
ParameterBlock multiRunParams = new
```

```
    ParameterBlock(StatisticalAnalyzerFactory.class.getName());
multiRunParams.addSubBl(StatisticalAnalyzerFactory.CONFIDENCE, 0.95);
multiRunParams.addSubBl(StatisticalAnalyzerFactory.ALLOWED_ERROR, 0.05);

//Evaluation for ranking according to performance
ParameterBlock evalParams = new
    ParameterBlock(ParetoRankingEvaluatorFactory.class.getName());
// define objectives, true means it should be minimized
Map<String, Boolean> objectives = new HashMap<>();
objectives.put("grossDataThrough", false);
objectives.put("nettoDataThrough", false);
objectives.put("maxDelayPacket", true);
objectives.put("avgDelayPacket", true);
objectives.put("maxDelayHeader", true);
objectives.put("avgDelayHeader", true);
objectives.put("avgDelayRouting", true);
objectives.put("recPackets", false);
objectives.put("recData", false);
evalParams.addSubBl(ParetoRankingEvaluatorFactory.OBJECTIVES, objectives);

//Assemble result parameter block
ParameterBlock specification = new ParameterBlock();
specification.addSubBl(BaseValidator.PROBLEM_URI, modelURI);
specification.addSubBl(BaseValidator.CONFIGURATOR, problemConfParams);
specification.addSubBl(BaseValidator.SIMULATOR_CONFIGS, simConfigs);
specification.addSubBl(BaseValidator.INSTRUMENTER, instConfigs);
specification.addSubBl(BaseValidator.MULTI_RUN_ANALYSIS, multiRunParams);
specification.addSubBl(BaseValidator.EVALUATION, evalParams);
```

C. Steady State Estimator Evaluation Results

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.7906	0.6667	0	0.3333
Schruben's Test	0.4345	0.7853	0.1574	0.0572
Moving Windows	1.7340	0.5625	0.1348	0.3026
Running Mean	1.7508	0.6597	0.0069	0.3333
MSER	1.0206	0.6206	0.1667	0.2126
Euclidean Distance	0.6393	0.7187	0.1177	0.1634
Balancing Mean	0.8776	0.5445	0.1221	0.3333
Goodness of Fit	0.0018	0.5	0.5	0
Stop Crossing Mean	1.8272	0.6667	0	0.3333
Crossing Mean	1.5804	0.5899	0.1351	0.274884259
Synthetic	0.14652801	0.884679643	0.0955873	0.019733057

Table C.1.: Overall results of the steady state estimator evaluation.

C.1. Steady State Estimator Evaluation Results with Different Noise Levels

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.82548	0.66667	0.0	0.33333
Schruben's Test	0.27657	0.79745	0.15856	0.04398
Moving Windows	1.81716	0.56019	0.13657	0.30324
Running Mean	1.82722	0.65972	0.00694	0.33333
MSER	3.18331	0.16667	0.66667	0.16667
Euclidean Distance	0.80043	0.80556	0.0	0.19444
Balancing Mean	1.56872	0.17824	0.48843	0.33333
Goodness of Fit	0.00181	1.0	0.0	0.0
Stop Crossing Mean	1.91038	0.66667	0.0	0.33333
Crossing Mean	1.45567	0.33333	0.44444	0.22222

Table C.2.: Steady state estimator evaluation with 0 percent noise in the tested time series.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.80582	0.66667	0.0	0.33333
Schruben's Test	0.27937	0.7963	0.15856	0.04514
Moving Windows	1.7729	0.56134	0.13657	0.30208
Running Mean	1.78264	0.65972	0.00694	0.33333
MSER	0.91914	0.77778	0.0	0.22222
Euclidean Distance	0.70554	0.80324	0.0	0.19676
Balancing Mean	0.78647	0.66667	0.0	0.33333
Goodness of Fit	NaN	0.33333	0.66667	0.0
Stop Crossing Mean	1.86341	0.66667	0.0	0.33333
Crossing Mean	1.32361	0.69444	0.05324	0.25231

Table C.3.: Steady state estimator evaluation with 5 percent noise in the tested time series.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.78586	0.66667	0.0	0.33333
Schruben's Test	0.55269	0.78009	0.15741	0.0625
Moving Windows	1.71846	0.56134	0.13426	0.3044
Running Mean	1.74026	0.65972	0.00694	0.33333
MSER	0.89527	0.77546	0.0	0.22454
Euclidean Distance	0.53064	0.70833	0.14005	0.15162
Balancing Mean	0.76536	0.66667	0.0	0.33333
Goodness of Fit	NaN	0.33333	0.66667	0.0
Stop Crossing Mean	1.81577	0.66667	0.0	0.33333
Crossing Mean	1.77778	0.66898	0.03125	0.29977

Table C.4.: Steady state estimator evaluation with 10 percent noise in the tested time series.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.74537	0.66667	0.0	0.33333
Schruben's Test	0.61376	0.76736	0.15509	0.07755
Moving Windows	1.62825	0.56713	0.13194	0.30093
Running Mean	1.65327	0.65972	0.00694	0.33333
MSER	0.84541	0.76273	0.0	0.23727
Euclidean Distance	0.36613	0.55787	0.33102	0.11111
Balancing Mean	0.72756	0.66667	0.0	0.33333
Goodness of Fit	NaN	0.33333	0.66667	0.0
Stop Crossing Mean	1.71931	0.66667	0.0	0.33333
Crossing Mean	1.67572	0.66319	0.01157	0.32523

Table C.5.: Steady state estimator evaluation with 20 percent noise in the tested time series.

C.2. Steady State Estimator Evaluation Results with Different Problem Lengths

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.23876	0.66667	0.0	0.33333
Schruben's Test	0.01695	0.81481	0.16667	0.01852
Moving Windows	0.18752	0.25	0.53935	0.21065
Running Mean	0.19374	0.63889	0.02778	0.33333
MSER	0.1289	0.61574	0.16667	0.21759
Euclidean Distance	0.11699	0.77778	0.0	0.22222
Balancing Mean	0.1317	0.55208	0.11458	0.33333
Goodness of Fit	0.00121	0.5	0.5	0.0
Stop Crossing Mean	0.2351	0.66667	0.0	0.33333
Crossing Mean	0.13829	0.60764	0.15046	0.2419

Table C.6.: Steady state estimator evaluation with time series comprising 100 data points.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.44378	0.66667	0.0	0.33333
Schruben's Test	0.19065	0.78935	0.15625	0.0544
Moving Windows	0.88432	0.66667	0.0	0.33333
Running Mean	1.1003	0.66667	0.0	0.33333
MSER	0.65791	0.62153	0.16667	0.21181
Euclidean Distance	0.66291	0.77546	0.0	0.22454
Balancing Mean	0.575	0.54282	0.12384	0.33333
Goodness of Fit	0.00156	0.5	0.5	0.0
Stop Crossing Mean	1.17888	0.66667	0.0	0.33333
Crossing Mean	0.90406	0.59144	0.13542	0.27315

Table C.7.: Steady state estimator evaluation with time series comprising 500 data points.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.95711	0.66667	0.0	0.33333
Schruben's Test	0.54125	0.77315	0.15278	0.07407
Moving Windows	1.84373	0.66667	0.0	0.33333
Running Mean	2.24746	0.66667	0.0	0.33333
MSER	1.32476	0.62384	0.16667	0.20949
Euclidean Distance	0.82882	0.66088	0.23727	0.10185
Balancing Mean	1.12536	0.54167	0.125	0.33333
Goodness of Fit	0.00202	0.5	0.5	0.0
Stop Crossing Mean	2.35717	0.66667	0.0	0.33333
Crossing Mean	2.01061	0.58102	0.13079	0.28819

Table C.8.: Steady state estimator evaluation with time series comprising 1000 data points.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	1.52289	0.66667	0.0	0.33333
Schruben's Test	0.92473	0.76389	0.15394	0.08218
Moving Windows	2.9969	0.66667	0.0	0.33333
Running Mean	3.41865	0.66667	0.0	0.33333
MSER	1.97935	0.62153	0.16667	0.21181
Euclidean Distance	1.27617	0.66088	0.2338	0.10532
Balancing Mean	1.6878	0.54167	0.125	0.33333
Goodness of Fit	0.00246	0.5	0.5	0.0
Stop Crossing Mean	3.53772	0.66667	0.0	0.33333
Crossing Mean	3.10947	0.57986	0.12384	0.2963

Table C.9.: Steady state estimator evaluation with time series comprising 1500 data points.

C.3. Steady State Estimator Evaluation Results with Different Bias Lengths

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.00087	1.0	0.0	0.0
Schruben's Test	0.00088	1.0	0.0	0.0
Moving Windows	0.00087	0.75	0.25	0.0
Running Mean	0.00087	1.0	0.0	0.0
MSER	0.00117	0.75	0.25	0.0
Euclidean Distance	0.0007	0.82465	0.17535	0.0
Balancing Mean	0.00116	0.75	0.25	0.0
Goodness of Fit	0.0	0.25	0.75	0.0
Stop Crossing Mean	0.00087	1.0	0.0	0.0
Crossing Mean	0.00117	0.75	0.25	0.0

Table C.10.: Steady state estimator evaluation with time series where the initial bias has a length of 0 percent of the time series length.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.01526	1.0	0.0	0.0
Schruben's Test	0.00089	0.52778	0.47222	0.0
Moving Windows	0.26406	0.84549	0.15451	0.0
Running Mean	0.38505	0.97917	0.02083	0.0
MSER	0.00117	0.75	0.25	0.0
Euclidean Distance	0.09937	0.82205	0.17795	0.0
Balancing Mean	0.26984	0.88368	0.11632	0.0
Goodness of Fit	0.00362	0.25	0.75	0.0
Stop Crossing Mean	0.41323	1.0	0.0	0.0
Crossing Mean	0.29424	0.84462	0.15538	0.0

Table C.11.: Steady state estimator evaluation with time series where the initial bias has a length of 50 percent of the time series length.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.44403	0.0	0.0	1.0
Schruben's Test	0.07227	0.99479	0.0	0.00521
Moving Windows	2.85315	0.09201	0.0	0.90799
Running Mean	3.11911	0.0	0.0	1.0
MSER	0.0288	0.64063	0.0	0.35938
Euclidean Distance	1.59939	0.51215	0.0	0.48785
Balancing Mean	0.10732	0.0	0.0	1.0
Goodness of Fit	NaN	1.0	0.0	0.0
Stop Crossing Mean	3.29601	0.0	0.0	1.0
Crossing Mean	2.77964	0.19097	0.0	0.80903

Table C.12.: Steady state estimator evaluation with time series where the initial bias has a length of 100 percent of the time series length.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	4.26749	0.0	0.0	1.0
Schruben's Test	4.35448	0.66146	0.0	0.33854
Moving Windows	6.21598	0.09201	0.0	0.90799
Running Mean	6.55722	0.0	0.0	1.0
MSER	4.7461	0.08333	0.0	0.91667
Euclidean Distance	3.62654	0.50694	0.0	0.49306
Balancing Mean	4.0369	0.0	0.0	1.0
Goodness of Fit	NaN	1.0	0.0	0.0
Stop Crossing Mean	6.83908	0.0	0.0	1.0
Crossing Mean	5.831	0.15972	0.0	0.84028

Table C.13.: Steady state estimator evaluation with time series where the initial bias has a length of 150 percent of the time series length.

C.4. Steady State Estimator Evaluation Results with Different Autocorrelation Factors

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.79089	0.66667	0.0	0.33333
Schruben's Test	0.43835	0.78472	0.15683	0.05845
Moving Windows	1.7327	0.56424	0.13368	0.30208
Running Mean	1.75165	0.65972	0.00694	0.33333
MSER	1.02	0.61979	0.16667	0.21354
Euclidean Distance	0.62218	0.69155	0.15336	0.15509
Balancing Mean	0.8799	0.54514	0.12153	0.33333
Goodness of Fit	0.00181	0.5	0.5	0.0
Stop Crossing Mean	1.8271	0.66667	0.0	0.33333
Crossing Mean	1.67046	0.58681	0.13137	0.28183

Table C.14.: Steady state estimator evaluation with time series comprising an induced autocorrelation with factor -0.5.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.79037	0.66667	0.0	0.33333
Schruben's Test	0.43077	0.78588	0.15799	0.05613
Moving Windows	1.73548	0.56076	0.136	0.30324
Running Mean	1.75005	0.65972	0.00694	0.33333
MSER	1.02135	0.62153	0.16667	0.21181
Euclidean Distance	0.65457	0.74595	0.08218	0.17188
Balancing Mean	0.87538	0.54398	0.12269	0.33333
Goodness of Fit	0.00181	0.5	0.5	0.0
Stop Crossing Mean	1.82733	0.66667	0.0	0.33333
Crossing Mean	1.48809	0.59317	0.13889	0.26794

Table C.15.: Steady state estimator evaluation with time series comprising an induced autocorrelation with factor 0.5.

C.5. Steady State Estimator Evaluation Results with Different Bias Trends

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	1.87307	0.5	0.0	0.5
Schruben's Test	1.77628	0.66406	0.25	0.08594
Moving Windows	3.91935	0.45833	0.08333	0.45833
Running Mean	4.20285	0.47917	0.02083	0.5
MSER	2.49768	0.5625	0.125	0.3125
Euclidean Distance	0.12215	0.77344	0.08854	0.13802
Balancing Mean	2.02887	0.40104	0.09896	0.5
Goodness of Fit	0.0021	0.625	0.375	0.0
Stop Crossing Mean	4.32622	0.5	0.0	0.5
Crossing Mean	3.74604	0.46875	0.11719	0.41406

Table C.16.: Steady state estimator evaluation with time series comprising quadratic bias trends with -100 percent slope.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	1.12332	0.5	0.0	0.5
Schruben's Test	1.08382	0.66667	0.25	0.08333
Moving Windows	2.35642	0.46094	0.08333	0.45573
Running Mean	2.52236	0.47917	0.02083	0.5
MSER	1.50505	0.5625	0.125	0.3125
Euclidean Distance	0.06354	0.77865	0.08854	0.13281
Balancing Mean	1.21835	0.39844	0.10156	0.5
Goodness of Fit	0.00126	0.625	0.375	0.0
Stop Crossing Mean	2.59587	0.5	0.0	0.5
Crossing Mean	2.2918	0.46875	0.11719	0.41406

Table C.17.: Steady state estimator evaluation with time series comprising quadratic bias trends with -60 percent slope.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.5612	0.5	0.0	0.5
Schruben's Test	0.51036	0.66406	0.25	0.08594
Moving Windows	1.19887	0.44531	0.09896	0.45573
Running Mean	1.26096	0.47917	0.02083	0.5
MSER	0.74733	0.55729	0.125	0.31771
Euclidean Distance	0.03668	0.76823	0.09375	0.13802
Balancing Mean	0.60846	0.39844	0.10156	0.5
Goodness of Fit	0.00063	0.625	0.375	0.0
Stop Crossing Mean	1.29868	0.5	0.0	0.5
Crossing Mean	1.1562	0.46094	0.125	0.41406

Table C.18.: Steady state estimator evaluation with time series comprising quadratic bias trends with -30 percent slope.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.56155	0.5	0.0	0.5
Schruben's Test	0.36936	0.82292	0.08333	0.09375
Moving Windows	1.23483	0.42969	0.11979	0.45052
Running Mean	1.23372	0.5	0.0	0.5
MSER	0.73311	0.54427	0.125	0.33073
Euclidean Distance	0.82233	0.55469	0.09115	0.35417
Balancing Mean	0.60822	0.40104	0.09896	0.5
Goodness of Fit	0.00063	0.625	0.375	0.0
Stop Crossing Mean	1.29606	0.5	0.0	0.5
Crossing Mean	1.16806	0.46875	0.11979	0.41146

Table C.19.: Steady state estimator evaluation with time series comprising quadratic bias trends with 30 percent slope.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	1.12237	0.5	0.0	0.5
Schruben's Test	0.7402	0.82292	0.08333	0.09375
Moving Windows	2.3824	0.46094	0.08854	0.45052
Running Mean	2.46843	0.5	0.0	0.5
MSER	1.48559	0.55469	0.125	0.32031
Euclidean Distance	1.64238	0.55729	0.08854	0.35417
Balancing Mean	1.20735	0.40104	0.09896	0.5
Goodness of Fit	0.00126	0.625	0.375	0.0
Stop Crossing Mean	2.5954	0.5	0.0	0.5
Crossing Mean	2.23833	0.47135	0.11719	0.41146

Table C.20.: Steady state estimator evaluation with time series comprising quadratic bias trends with 60 percent slope.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	1.86847	0.5	0.0	0.5
Schruben's Test	1.05763	0.84375	0.08333	0.07292
Moving Windows	3.96122	0.45833	0.08854	0.45313
Running Mean	4.11559	0.5	0.0	0.5
MSER	2.47966	0.55469	0.125	0.32031
Euclidean Distance	2.72705	0.55729	0.08854	0.35417
Balancing Mean	2.02038	0.40104	0.09896	0.5
Goodness of Fit	0.0021	0.625	0.375	0.0
Stop Crossing Mean	4.32193	0.5	0.0	0.5
Crossing Mean	3.84677	0.47135	0.11979	0.40885

Table C.21.: Steady state estimator evaluation with time series comprising quadratic bias trends with 100 percent slope.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.00413	1.0	0.0	0.0
Schruben's Test	0.00091	0.54167	0.45833	0.0
Moving Windows	0.00453	0.78646	0.21354	0.0
Running Mean	0.00536	1.0	0.0	0.0
MSER	0.00117	0.75	0.25	0.0
Euclidean Distance	0.00074	0.82813	0.17188	0.0
Balancing Mean	0.0053	0.83333	0.16667	0.0
Goodness of Fit	0.00362	0.25	0.75	0.0
Stop Crossing Mean	0.00564	1.0	0.0	0.0
Crossing Mean	0.00569	0.83333	0.16667	0.0

Table C.22.: Steady state estimator evaluation with time series comprising constant bias trends with level -100 percent.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.00158	1.0	0.0	0.0
Schruben's Test	0.00088	0.83333	0.16667	0.0
Moving Windows	0.003	0.83333	0.16667	0.0
Running Mean	0.00366	1.0	0.0	0.0
MSER	0.00116	0.75	0.25	0.0
Euclidean Distance	0.0011	0.82292	0.17708	0.0
Balancing Mean	0.00357	0.83333	0.16667	0.0
Goodness of Fit	0.00217	0.25	0.75	0.0
Stop Crossing Mean	0.00373	1.0	0.0	0.0
Crossing Mean	0.00381	0.83333	0.16667	0.0

Table C.23.: Steady state estimator evaluation with time series comprising constant bias trends with level -60 percent.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.00121	1.0	0.0	0.0
Schruben's Test	$8.7 \cdot 10^{-}4$	1.0	0.0	0.0
Moving Windows	0.00215	0.81771	0.18229	0.0
Running Mean	0.00228	1.0	0.0	0.0
MSER	0.00117	0.75	0.25	0.0
Euclidean Distance	0.00182	0.81771	0.18229	0.0
Balancing Mean	0.00229	0.83333	0.16667	0.0
Goodness of Fit	0.00109	0.25	0.75	0.0
Stop Crossing Mean	0.0023	1.0	0.0	0.0
Crossing Mean	0.00243	0.83333	0.16667	0.0

Table C.24.: Steady state estimator evaluation with time series comprising constant bias trends with level -30 percent.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.00108	1.0	0.0	0.0
Schruben's Test	0.00091	1.0	0.0	0.0
Moving Windows	0.00158	0.76042	0.23958	0.0
Running Mean	0.00173	1.0	0.0	0.0
MSER	0.00117	0.75	0.25	0.0
Euclidean Distance	0.00141	0.83333	0.16667	0.0
Balancing Mean	0.0017	0.83333	0.16667	0.0
Goodness of Fit	0.00109	0.25	0.75	0.0
Stop Crossing Mean	0.00172	1.0	0.0	0.0
Crossing Mean	0.00181	0.83333	0.16667	0.0

Table C.25.: Steady state estimator evaluation with time series comprising constant bias trends with level 30 percent.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.00148	1.0	0.0	0.0
Schruben's Test	0.00089	0.95833	0.04167	0.0
Moving Windows	0.00283	0.75	0.25	0.0
Running Mean	0.00319	1.0	0.0	0.0
MSER	0.00116	0.75	0.25	0.0
Euclidean Distance	0.00257	0.83333	0.16667	0.0
Balancing Mean	0.003	0.83333	0.16667	0.0
Goodness of Fit	0.00217	0.25	0.75	0.0
Stop Crossing Mean	0.00315	1.0	0.0	0.0
Crossing Mean	0.00328	0.83333	0.16667	0.0

Table C.26.: Steady state estimator evaluation with time series comprising constant bias trends with level 60 percent.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.00193	1.0	0.0	0.0
Schruben's Test	0.00087	0.83333	0.16667	0.0
Moving Windows	0.00433	0.75	0.25	0.0
Running Mean	0.00514	1.0	0.0	0.0
MSER	0.00117	0.75	0.25	0.0
Euclidean Distance	0.00407	0.82292	0.17708	0.0
Balancing Mean	0.00471	0.83333	0.16667	0.0
Goodness of Fit	0.00362	0.25	0.75	0.0
Stop Crossing Mean	0.00505	1.0	0.0	0.0
Crossing Mean	0.00524	0.83333	0.16667	0.0

Table C.27.: Steady state estimator evaluation with time series comprising constant bias trends with level 100 percent.

C.6. Steady State Estimator Evaluation Results with Different Bias Shapes

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.66489	0.66667	0.0	0.33333
Schruben's Test	0.00089	0.77083	0.22917	0.0
Moving Windows	0.17624	0.53819	0.12847	0.33333
Running Mean	0.11979	0.66667	0.0	0.33333
MSER	1.29581	0.6658	0.16667	0.16753
Euclidean Distance	0.11908	0.61285	0.11892	0.26823
Balancing Mean	0.89142	0.5	0.16667	0.33333
Goodness of Fit	0.00082	0.5	0.5	0.0
Stop Crossing Mean	0.32678	0.66667	0.0	0.33333
Crossing Mean	0.15635	0.58333	0.08333	0.33333

Table C.28.: Steady state estimator evaluation with time series comprising oscillating bias shapes.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	0.57343	0.66667	0.0	0.33333
Schruben's Test	0.63603	0.76128	0.10417	0.13455
Moving Windows	1.70007	0.55903	0.11632	0.32465
Running Mean	1.68781	0.66667	0.0	0.33333
MSER	0.78669	0.5	0.16667	0.33333
Euclidean Distance	0.25003	0.79861	0.11632	0.08507
Balancing Mean	0.52496	0.55035	0.11632	0.33333
Goodness of Fit	0.00056	0.5	0.5	0.0
Stop Crossing Mean	1.71881	0.66667	0.0	0.33333
Crossing Mean	1.86573	0.58333	0.08333	0.33333

Table C.29.: Steady state estimator evaluation with time series comprising random bias shapes.

Steady State Estimator	Mean Deviation	Success Rate	False Negatives	False Positives
Batch Means	1.13358	0.66667	0.0	0.33333
Schruben's Test	0.52183	0.82378	0.13889	0.03733
Moving Windows	3.56707	0.59028	0.15972	0.25
Running Mean	3.48099	0.64583	0.02083	0.33333
MSER	1.03845	0.69618	0.16667	0.13715
Euclidean Distance	1.61922	0.74479	0.11806	0.13715
Balancing Mean	1.20512	0.58333	0.08333	0.33333
Goodness of Fit	0.00406	0.5	0.5	0.0
Stop Crossing Mean	3.43607	0.66667	0.0	0.33333
Crossing Mean	3.36223	0.6033	0.23872	0.15799

Table C.30.: Steady state estimator evaluation with time series comprising sharp bias shapes.

C.7. Decision Tree Trained for Steady State Estimation

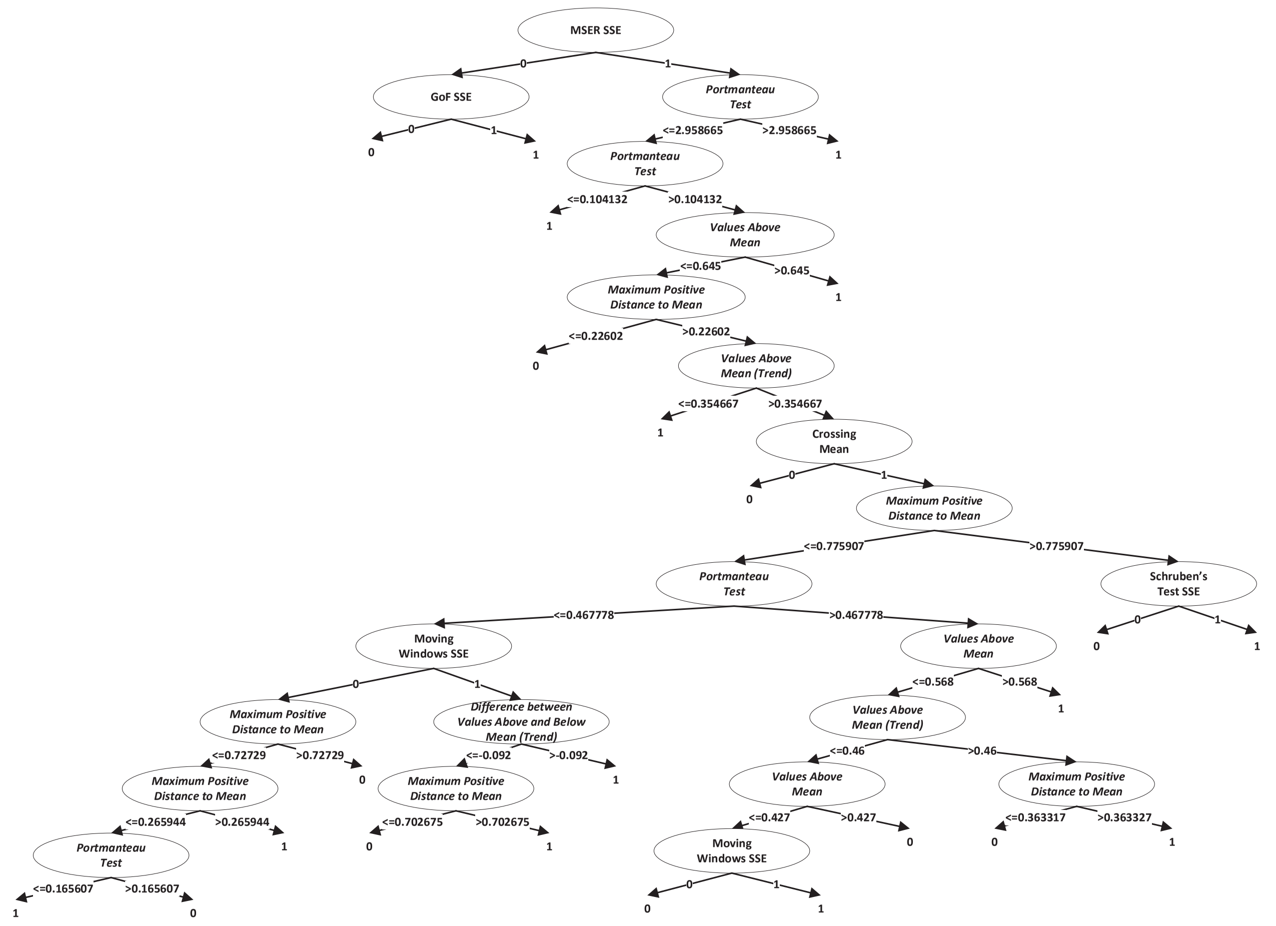


Figure C.1.: Decision tree of the synthetic steady state estimator. A decision of 0 denotes that the end of warm-up phase has not been reached in the time series under investigation, a result of 1 denotes that it has been reached.

C.8. Steady State Estimator Evaluation Results with Simulation Output Data

	MgCl2		NaCl		kNa2Cl	
Steady State Estimator	Deviation	Chunks	Deviation	Chunks	Deviation	Chunks
Batch Means	0.16678	-2.23	0.04820	1.25	0.05450	-0.91
Schruben's Test	0.04095	1.11	0.03342	2.74	0.04798	-0.75
Moving Windows	0.00434	3.48	0.01112	3.93	0.01301	3.35
Running Mean	NaN	10	0.05253	1.85	0.06355	2.94
MSER	0.30774	-3.82	0.06069	0.68	0.08334	-1.72
Euclidean Distance	0.46379	6.7	NaN	10	NaN	10
Balancing Mean	0.23564	-3	0.03673	1.92	0.06023	-0.97
Goodness of Fit	0.13081	1.3	NaN	10	0.09618	5.88
Stop Crossing Mean	0.07449	9.96	NaN	10	NaN	10
Crossing Mean	NaN	10	NaN	10	NaN	10
Synthetic	0.00280	2.62	0.00711	2.43	0.01014	1.55

	HCl		rNHCOR		Mapk		TCR	
Steady State Estimator	Deviation	Chunks	Deviation	Chunks	Deviation	Chunks	Deviation	Chunks
Batch Means	0.00006	0.03	NaN	10	0.04565	-2	0.27655	3
Schruben's Test	0.00382	0.99	NaN	10	0.05388	-2.03	0.94547	-1
Moving Windows	0.00232	3	NaN	10	0.03451	-1.28	NaN	10
Running Mean	0.00447	-0.05	NaN	10	NaN	10	NaN	10
MSER	0.00025	0.07	NaN	10	0.04711	-1.7	0.94547	-1
Euclidean Distance	NaN	10	NaN	10	NaN	10	NaN	10
Balancing Mean	0.00206	0.15	NaN	10	0.08645	-2.18	NaN	10
Goodness of Fit	NaN	10	NaN	10	0.31742	9.74	NaN	10
Stop Crossing Mean	NaN	10	NaN	10	NaN	10	NaN	10
Crossing Mean	NaN	10	NaN	10	NaN	10	NaN	10
Synthetic	0.00099	0.29	NaN	10	0.01489	0.82	0.05453	5

Table C.31.: Mean results of the steady state estimator evaluation with simulation output data.

D. Statistical Analyzer Evaluation Results

D.1. Statistical Analyzer Evaluation Results with Training Data

Statistical Analyzer	Mean Sample Size Deviation	Mean Relative Iterations
Iterative Normal	0.0043534	0.010395
Iterative Student-T	0.0039214	0.011943
Two-stepped Normal	0.084126	0.000741
Two-Stepped Student-T	0.049621	0.000311
Synthetic (Replication Optimized)	0.0040277	0.011022
Synthetic (Iteration Optimized)	0.049621	0.000311

Table D.1.: Statistical analyzer evaluation with training data.

D.2. Statistical Analyzer Evaluation Results with Simulation Output Data

	MgCl2		NaCl	
Statistical Analyzer	Mean Sample Size Deviation	Mean Relative Iterations	Mean Sample Size Deviation	Mean Relative Iterations
Iterative Normal	0.036995516	0.010369955	0.019690577	0.010196906
Iterative Student-T	0.00896861	0.010089686	0.019690577	0.010196906
Two-stepped Normal	0.051849776	0.000560538	0.08790436	0.000703235
Two-Stepped Student-T	0.065302691	0.000840807	0.051687764	0.000703235
Synthetic (Replication Optimized)	0.00896861	0.010089686	0.019690577	0.010196906
Synthetic (Iteration Optimized)	0.009248879	0.000840807	0.01652602	0.000703235
	kNa2Cl		HCl	
Statistical Analyzer	Mean Sample Size Deviation	Mean Relative Iterations	Mean Sample Size Deviation	Mean Relative Iterations
Iterative Normal	0.007506914	0.010075069	0.014070224	0.010140702
Iterative Student-T	0.007506914	0.010075069	0.001394347	0.010013943
Two-stepped Normal	0.055314105	0.000395101	0.02699962	0.000253518
Two-Stepped Student-T	0.033188463	0.000395101	0.060590696	0.000253518
Synthetic (Replication Optimized)	0.007506914	0.010075069	0.001394347	0.010013943
Synthetic (Iteration Optimized)	0.033188463	0.000395101	0.02699962	0.000253518
	rNHCOR		Mapk	
Statistical Analyzer	Mean Sample Size Deviation	Mean Relative Iterations	Mean Sample Size Deviation	Mean Relative Iterations
Iterative Normal	0.006253145	0.010062531	0.005698961	0.01005699
Iterative Student-T	0.006253145	0.010062531	0.005698961	0.01005699
Two-stepped Normal	0.046215769	0.00014375	0.091350989	0.000251425
Two-Stepped Student-T	0.029612593	0.00014375	0.078360711	0.000167616
Synthetic (Replication Optimized)	0.006253145	0.010062531	0.005698961	0.01005699
Synthetic (Iteration Optimized)	0.029612593	0.00014375	0.078360711	0.000167616
	TCR			
Statistical Analyzer	Mean Sample Size Deviation	Mean Relative Iterations		
Iterative Normal	0.000648569	0.010006486		
Iterative Student-T	0.000648569	0.010006486		
Two-stepped Normal	0.00403039	0.000138979		
Two-Stepped Student-T	0.086305939	9.26526E-05		
Synthetic (Replication Optimized)	0.000648569	0.010006486		
Synthetic (Iteration Optimized)	0.00403039	0.000138979		

Table D.2.: Statistical analyzer evaluation with simulation data.

Bibliography

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity in sequence databases. In *Proceedings of the International Conference on Foundations of Data Organization and Algorithms (FODO)*, pages 69–84, 1993.

[2] K. Al-Zoubi and G. Wainer. Managing simulation workflow patterns using dynamic serviecoriented compositions. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 765–777, 2010.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 423–424, 2004.

[4] L. Andersen. JDBC API specification 4.1. Technical report, Oracle America Inc., 2011.

[5] A. Angermann, M. Beuschel, M. Rau, and U. Wohlfarth. *MATLAB- Simulink- Stateflow*. Oldenbourg, 2009.

[6] D. Ashlock. *Evolutionary cmputation for modeling and optimization*. Springer, 2006.

[7] S. Asmussen and P. W. Glynn. *Stochastic simulation*. Springer, 2007.

[8] S. Asmussen, P. W. Glynn, and H. Thorisson. Stationarity detection in the initial transient problem. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2(2):130–157, 1992.

[9] P. W. Atkins. *Physical chemistry*. W.H. Freeman & Company, 9th edition, 1997.

[10] A. M. Law Averill and D. M. Kelton. *Simulation modeling and analysis*. McGraw-Hill International, 4th edition, 2007.

[11] B. Baffour and Paolo Valente. Census quality evaluation: considerations from an international perspective. In *Proceedings of the Joint UNECE/Eurostat Meeting on Population and Housing Censuses*, pages 1–22, 2008.

[12] O. Balci. Guidelines for successful simluation studies (tutorial session). In *Proceedings of the Winter Simulation Conference (WSC)*, pages 25–32, 1990.

[13] O. Balci. Verification, validation, and accreditation. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 41–48, 1998.

[14] R. Barga and D. Gannon. Scientific versus business workflows. *Workflows for eScience*, pages 9–16, 2007.

[15] M. Barnasconi. Viewpoint: analog/mixed-signal. Technical report, Open SystemC Initiative, 2009.

[16] R. R. Barton. Designing simulation experiments. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 73–79, 2004.

[17] B. Beckman. Why LINQ matters: cloud composability guaranteed. *Communications of the ACM*, 55(4):38–44, April 2012.

[18] L. Benini and G. De Micheli. Networks on chips: a new SoC paradigm. *Computer*, 35(1):70–78, 2002.

[19] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): generic syntax. Technical report, Network Working Group, 2005.

[20] C. M. Bishop. *Neural networks for pattern recognition.* Oxford University Press, 1995.

[21] A. Boukerche. An Adaptive Partitioning Algorithm for Conservative Parallel Simulation. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1353–1358, 2001.

[22] S. Bowers, B. Ludascher, A. H. H. Ngu, and T. Critchlow. Enabling scientific workflow reuse through structured composition of dataflow and control-flow. In *Proceedings of the International Conference on Data Engineering Workshops (ICDEW)*, pages 70–80, 2006.

[23] G. E. P. Box and D. A. Pierce. Distribution of residual correlations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 65:1509–1526, 1970.

[24] P. Bratley, B. L. Fox, and L. E. Schrage. *A guide to simulation.* Springer, 1987.

[25] W. Bryc. *The normal distribution: characterizations with applications.* Springer, 1995.

[26] W. B. Cannon. Organization for physiological homeostasis. *Journal of Physiolical Reviews*, 9:399–431, 1929.

[27] C. R. Cash, D. G. Dippold, J. M. Long, and B. L. Nelson. Evaluation of tests for initial-condition bias. In *Proceddings of the Winter Simulation Conference (WSC)*, pages 577–585, 1992.

[28] G. Castelo-Branco, J. Wagner, F. J. Rodriguez, J. Kele, K. Sousa, N. Rawal, H. Amalia Pasolli, E. Fuchs, J. Kitajewski, and E. Arenas. Differential regulation of midbrain dopaminergic neuron development by wnt-1, wnt-3a, and wnt-5a. *Proceedings of the National Academy of Sciences*, 100(22):12747–12752, 2003.

[29] F. E. Cellier. *Continuous system modeling.* Springer, 1991.

[30] F. E. Cellier and E. Kofman. *Continuous system simulation.* Springer, 2006.

[31] A. Cimino, F. Longo, and G. Mirabelli. A general simulation framework for supply chain modeling: state of the art and case study. *International Journal of Computer Science*, 7(3):1–9, March 2010.

[32] The Workflow Management Coalition. Workflow management coalition workflow standard process definition interface — XML process definition language, 2008.

[33] K. Concannon, M. Elder, K. Hunter, J. Tremble, and S. Tse. *Simulation modeling with SIMUL8.* Visual Thinking International Ltd., 2003.

[34] Geer Mountain Software Corporation. Stat::Fit, accessed 5/2013. http://www.geerms.com/index.htm.

[35] R. J. Dakin. A tree-search algorithm for mixed integer programming problems. *The Computer Journal*, 8:250–255, 1965.

[36] H. Daly. *Steady state economics.* Island Press, 2nd edition, 1991.

[37] S. R. Das. Adaptive protocols for parallel discrete event simulation. *Proceedings of the Winter Simulation Conference (WSC)*, pages 186–193, 1996.

[38] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist Non-dominated Sorting Genetic Algorithm for multi-objective optimization: NSGA-II. In *Parallel Problem Solving from Nature (PPSN)*, volume 1917, pages 849–858. Springer, 2000.

[39] L. Dematté, R. Larcher, A. Palmisano, C. Priami, and A. Romanel. Programming biology in BlenX. *Systems Biology for Signaling Networks*, 1:777–820, 2010.

[40] L. Dematté, C. Priami, and A. Romanel. The Beta Workbench: a computational tool to study the dynamics of biological systems. *Briefings in Bioinformatics*, 9(5):437–449, 2008.

[41] E. Deza and M. M. Deza. *Encyclopedia of distances.* Springer, 2009.

[42] R. Donaldson and D. Gilbert. A model checking approach to the parameter estimation of biochemical pathways. In *Formal Methods for Computational Systems Biology*, pages 269–287, 2008.

[43] T. Dreibholz, E. P .Rathgeb, and X. Zhou. SimProcTC: the design and realization of a powerful tool-chain for OMNeT++ simulations. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools)*, pages 1–8, 2009.

[44] C. Ellis and K. Keddara. ML-DEWS: Modeling language to support dynamic evolution within workflow systems. *Journal of Computer Supported Cooperative Work*, 9(3–4):293–333, 2000.

[45] J. R. Emshoff and R. L. Sisson. *Design and use of computer simulation models.* The MacMillan Company, 1970.

[46] D. L. Ermak and J. A. McCammon. Brownian dynamics with hydrodynamics interactions. *Journal of Chemical Physics*, 69:1352–1360, 1978.

[47] L. Evans and P. Bryant. LHC machine. *Journal of Instrumentation*, 3(8):1–158, 2008.

[48] R. Ewald. *Automatic algorithm selection for complex simulation problems.* Phd thesis, Faculty of Computer Science and Electrical Engineering, University of Rostock., 2010.

[49] R. Ewald, J. Himmelspach, and A. M. Uhrmacher. A non-fragmenting partitioning algorithm for hierarchical models. In *Proceedings of the Winter Simulation Conference*, pages 848–855, 2006.

[50] R. Ewald and A. M. Uhrmacher. Setting up simulation wxperiments with SESSL. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 379:1–379:2, 2012.

[51] G. Ewing, K. Pawlikowski, and D. McNickle. Akaroa-2: Exploiting network computing by distributing stochastic simulation. In *Proceedings of the European Simulation Multi-Conference (ESM)*, pages 175–181, 1999.

[52] F. Fages, D. Jovanovska, A. Rizk, and S. Soliman. BioCham 3.2 reference manual. Technical report, INRIA Paris-Rocquencourt, 2010.

[53] F. Fages and A. Rizk. On the analysis of numerical data time series in temporal logic. In *Proceedings of the Conference on Computational Methods in Systems Biology (CMSB)*, pages 48–63, 2007.

[54] A. Ferscha. Probabilistic adaptive direct optimism control in time warp. In *Proceedings of the Workshop on Parallel and Distributed Simulation (PADS)*, pages 120–129, Washington, DC, USA, 1995.

[55] Frank Figge. Bio-folio: applying portfolio theory to biodiversity. *Biodiversity & Conservation*, 13(4):827–849, 2004.

[56] B. Franklin. Experiments and observations on electricity, made at Philadelphia in America. *Philosophical Transactions of the Royal Society*, 1751–1752.

[57] M. F. Franklin and R. A. Bailey. Selecting defining contrasts and confounded effects in p^{n-m} experiments. *Technometrics*, 27:165–172, 1977.

[58] T. L. Friesz, R. L. Tobin, H.-J. Cho, and N. J. Mehta. Sensitivity analysis based heuristic algorithms for mathematical programs with variational inequality constraints. *Mathematical Programming*, 48(1-3):265–284, 1990.

[59] J. S. Hunter G. E. Box, W. G. Hunter. *Statistics for experimenters: design, innovation, and discovery.* Wiley, 2nd edition, 2005.

[60] A. V. Gafarian, C. J. Ancker, and T. Morisaku. The problem of the initial transient in digital computer simulation. In *Proceddings of the Winter Simulation Conference (WSC)*, pages 49–51, 1976.

[61] A. V. Gafarian, C. J. Ancker, and T. Morisaku. Evaluation of commonly used rules for detecting steady state. *Computer Simulation*, 25(5):511–529, 1978.

[62] M. Gagliolo, V. Zhumatiy, and J. Schmidhuber. Adaptive online time allocation to search algorithms. Technical report, Istituto Dalle Molle di studi sull'intelligenza artificiale, 2004.

[63] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[64] S. Ghosh, Y. Matsuoka, Y. Asai, K.-Y. Hsin, and H. Kitano. Software for systems biology: from tools to integrated platforms. *Nature Reviews Genetics*, 12:821–832, 2011.

[65] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *Journal of Physical Chemistry*, 104:1876–1889, 2000.

[66] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. C. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the Challenges of Scientific Workflows. *Computer*, 40(12):24–32, 2007.

[67] D. T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of Physical Chemistry*, 22:403–434, 1976.

[68] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81:2340–2361, 1977.

[69] F. Glover. Tabu search — part 1. *INFORMS Journal on Computing*, 1(2):190–206, 1989.

[70] F. Glover. Tabu search — part 2. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

[71] F. Glover, J. P. Kelly, and M. Laguna. The OptQuest callable library user's documentation. Technical report, Optimization Technologies, Inc, 1999.

[72] C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1–2):43–62, February 2001.

[73] P. Goodwin. The Holt-Winters approach to exponential smoothing: 50 years old and going strong. *Foresight: The International Journal of Applied Forecasting*, pages 30–33, 2010.

[74] Rob Gordon. *Essential JNI: Java Native Interface.* Prentice Hall, 1998.

[75] J. Gosling, B. Joy, G. L. Steele Jr., G. Bracha, and A. Buckley. *The Java language specification, Java se 7 edition (Java series).* Oracle, 2013.

[76] F. Haack, S. Leye, and A. M. Uhrmacher. A flexible architecture for modeling and simulation of diffusional association. In *Workshop From Biology To Concurrency and back (FBTC)*, pages 70–84, 2010.

[77] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11:10–18, 2009.

[78] Houghton Mifflin Harcourt. *The American Heritage Science Dictionary.* Houghton Mifflin Company, 2005.

[79] T. Helms, R. Ewald, S. Rybacki, and A. M. Uhrmacher. A generic adaptive simulation algorithm for component-based simulation systems. In *Proceedings of the International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 1–10, 2013.

[80] T. Helms, J. Himmelspach, C. Maus, O. Röwer, J. Schützel, and A. M. Uhrmacher. Toward a language for the flexible observation of simulations. In *Proceedings of the Winter Simulation Conference*, pages 1–12, 2012.

[81] J. Himmelspach. *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations- und Experimentiersystems.* Phd thesis, University of Rostock, 2007.

[82] J. Himmelspach. Tutorial on building M&S software based on reuse. In *Proceedings of the Winter Simulation Conference (WSC)*, page 167, 2012.

[83] J. Himmelspach, R. Ewald, and A. M. Uhrmacher. A flexible and scalable experimentation layer. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 827–835, 2008.

[84] J. Himmelspach and A. M. Uhrmacher. Sequential processing of PDEVS models. In *Proceedings of the European Modeling and Simulation Symposium (EMSS)*, pages 239–244, Barcelona, Spain, 2006. ISBN 84-690-0726-2.

[85] J. Himmelspach and A. M. Uhrmacher. Plug'n simulate. In *Spring Simulation Multiconference*, pages 137–143, March 2007.

[86] V. Hlupic. Discrete-event simulation software: what the users want. *Journal of Simulation*, 73(6):362–370, 1999.

[87] K. Hoad, S. Robinson, and R. Davies. Automating warm-up length estimation. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 532–540, 2008.

[88] K. Hoad, S. Robinson, and R. Davies. AutoSimOA: a framework for automated analysis of simulation output. *Journal of Simulation*, 5:9–24, 2011.

[89] B. W. Hollocks. Discrete-event simulation: an inquiry into user practice. *Simulation Practice and Theory*, 8:451–471, 2001.

[90] R. Hooke and T.A. Jeeves. "direct search" solution of numerical and statistical problems. *Journal of the Association for Computing Machinery (ACM)*, 8(2):212–229, 1961.

[91] J. N. Hooker. Needed: an empirical science of algorithms. *Operations Research*, 42(2):201–212, 1994.

[92] S. Hoops, S. Sahle, R. Gauges, C. Lee, J. Pahle, N. Simus, M. Singhal, L. Xu, P. Mendes, and U. Kummer. COPASI–a COmplex PAthway SImulator. *Bioinformatics*, 22(24):3067–3074, 2006.

[93] 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 (TOMS)*, 26(2):227–253, 2000.

[94] W. H. Hsu, R. Joehanes, J. A. Louis, J. W. Plummer, and C. P. Schmidt. Machine learning in Java, accessed 5/2013. http://sourceforge.net/projects/mldev/.

[95] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.

[96] M. Hucka, A. Finney, H. Sauro, B. Kovitz, S. Keating, J. Matthews, and H. Bolouri. Introduction to the Systems Biology Workbench. Technical report, Systems Biology Workbench Development Group, California Institute of Technology, 2003.

[97] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, 2006.

[98] IBM. Autonomic computing: IBM's perspective on the state of information technology, 2001. `www.research.ibm.com/autonomic/manifesto/autonomic\_computing.pdf`.

[99] A. Indrayan. Elements of medical research. *Indian Journal of Medical Research (IJMR)*, 119(3):93–100, 2004.

[100] ISO - International Organization for Standardization. ISO 9001:2008 — quality management systems — requirements, 2008.

[101] J. A. Jacko. *Human-computer interaction handbook.* CRC Press, 3rd edition, 2012.

[102] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri nets and CPN tools for modeling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3–4):213–254, 2007.

[103] M. Jeschke. *Efficient non-spatial and spatial simulation of biochemical reaction networks.* Phd thesis, Universität Rostock, Rostock, 2010.

[104] M. Jeschke, R. Ewald, and A. M. Uhrmacher. Exploring the performance of spatial stochastic simulation algorithms. *Journal of Computational Physics*, 230(7):2562–2574, 2011.

[105] M. John, R. Ewald, and A. M. Uhrmacher. A spatial extension to the pi calculus. In *Electronic Notes in Theoretical Computer Science*, volume 194, pages 133–148, Amsterdam, The Netherlands, 2008.

[106] M. B. Juric. *Business process execution language for web services BPEL and BPEL4WS.* Packt Publishing, 2nd edition, January 2006.

[107] S. Hinz K., Schmidt, and C. Stahl. Transforming BPEL to Petri nets. In *Proceedings of the International Conference on Business Process Management (BPM)*, volume 3649, pages 220–235, 2005.

[108] D. W. Kelton, R. Sadowski, and D. Sturrock. *Simulation with Arena.* McGraw-Hill Professional, 5th edition, 2006.

[109] W. D. Kelton. Experimental design for simulation. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 32–38, 2000.

[110] J. Kennedy and R. Eberhart. Particle swarm optimization. *International Conference on Neural Networks (IJCNN)*, 4:1942–1948, 1995.

[111] M. Kima, S. H. Yoonb, P. A. Domanskib, and W. V. Payneb. Design of a steady-state detector for fault detection and diagnosis of a residential air conditioner. *International Journal of Refrigeration*, 31:791–792, 2007.

[112] G. King, C. Bauer, S. Ebersole, M. Rydahl Andersen, E. Bernard, H. Ferentschik, A. Warski, and G. Badner. Hibernate developer guide. Technical report, Red Hat, Inc., 2011.

[113] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[114] H. Kitano. Violations of robustness trade-offs. *Molecular Systems Biology*, 6(384), 2010.

[115] J. P. C. Kleijnen. *Design and analysis of simulation experiments.* Springer, 2008.

[116] J. P. C. Kleijnen. Design of experiments: overview. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 479–488, 2008.

[117] G. A. Korn and J. V. Wait. *Digital continuous-system simulation.* Prentice Hall, 1978.

[118] S. Kurkowski. *Credible mobile ad hoc network simulation-based studies.* Phd thesis, Colorado School of Mines, 2006.

[119] E. Lee, A. Salic, R. Krüger, H. Reinhart, and M. W. Kirschner. The roles of APC and axin derived from experimental and theoretical analysis of the wnt pathway. *Public Library of Science (PLoS) Biology*, 1(1):116–132, 2003.

[120] Y.-H. Lee, K.-H. Kyung, and C.-S. Jung. On-line determination of steady state in simulation outputs. *Computers industrial engineering*, 33(3):805–808, 1997.

[121] S. Leye, J. Himmelspach, and A. M. Uhrmacher. A discussion on experimental model validation. In *Proceedings of the International Conference on Computer Modelling and Simulation (UKSIM)*, pages 161–167, 2009.

[122] S. Leye, M. John, and A. M. Uhrmacher. A flexible architecture for performance experiments with the pi-calculus and its extensions. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools)*, pages 35:1–35:10, 2010.

[123] S. Leye, O. Mazemondet, and A. M. Uhrmacher. Parallel analysis with FAMVal to speed up simulation-based model checking. In *European Symposium on Computer Modeling and Simulation (EMS)*, pages 344–350, 2010.

[124] S. Leye, C. Priami, and A. M. Uhrmacher. A bounded-optimistic, parallel beta-binders simulator. In *Proceedings of the International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)*, pages 139–148, 2008.

[125] S. Leye and A. M. Uhrmacher. A flexible and extensible architecture for experimental model validation. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools)*, pages 65:1–65:10, 2010.

[126] S. Leye and A. M. Uhrmacher. GUISE - a tool for GUIding Simulation Experiments. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 305:1–305:2, 2012.

[127] B. Lustig, B. Jerchow, M. Sachs, S. Weiler, T. Pietsch, U. Karsten, M. van de Wetering, H. Clevers, P. M. Schlag, W. Birchmeier, and J. Behrens. Negative feedback loop of wnt signaling through upregulation of conductin/axin2 in colorectal and liver tumors. *Molecular and Cellular Biology*, 22:1184–1193, 2002.

[128] H. Markowitz. Portfolio selection. *Journal of Finance*, 7(1):77–91, 1952.

[129] P. Marrone. *Java object oriented neural engine: the complete guide.* HQBooks, 2007.

[130] M. Matsumoto and T. Nishimura. Mersenne twister. a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[131] O. Mazemondet, M. John, S. Leye, A. Rolfs, and A. M. Uhrmacher. Elucidating the sources of β-catenin dynamics in human neural progenitor cells. *Public Library of Science (PLoS) ONE*, 7(8):1–12, 2012.

[132] M. D. McKay and W. J. Conover R. J. Beckman. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *ASA Journal of Technometrics*, 21(2):239–245, 1979.

[133] P. K. Mckinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[134] S. Meraji, C. Tropper, and W. Zang. A multi-state q-learning approach for the dynamic load balancing of time warp. In *Proceedings of the Workshop on Principles of Advanced and Distributes Simulation (PADS)*, pages 1–8, 2010.

[135] E. Millman, D. Arora, and S. W. Neville. STARS: a framework for statistically rigorous simulation-based network research. In *Proceedings of the International Conference on Advanced Information Networking and Applications (WAINA)*, pages 733–739, 2011.

[136] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part i and ii. *Information and Computation*, 96:1–77, 1992.

[137] M. Minsky. Models, minds, machines. In *Proceedings of the International Federation for Information Processing (IFIP) Congress*, pages 45–49, 1965.

[138] R. Minson and G. Theodoropoulos. Adaptive support of range queries via push-pull algorithms. In *Proceedings of the International Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 53–60, 2007.

[139] T. Murata. Petri nets: properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[140] S. H. Northrup, S. A. Allison, and J. A. McCammon. Brownian dynamics simulation of diffusion-influenced bimolecular reactions. *Journal of Chemical Physics*, 80:1517–1524, 1984.

[141] P. Norvig and D. Cohn. Adaptive software. *PC AI Magazine*, 11(1):27–30, 1997.

[142] NS-3 Consortium. NS-3 manual. Technical report, NS-3 Project, 2012.

[143] T. I. Ören and B. P. Zeigler. Concepts for advanced simulation methodologies. *Simulation*, 32(3):69–82, 1979.

[144] K. Pawlikowski. Steady-state simulation of queueing processes: a survey of problems and solutions. *Computing Surveys*, 122(2):123–170, 1990.

[145] K. Pawlikowski. Towards credible and fast quantitative stochastic simulation. In *Proceedings of the International Conference on Design Analysis and Simulation of Distributed Systems*, pages 1–8, 2003.

[146] 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.

[147] L. F. Perrone, C. Cicconetti, G. Stea, and B. C. Ward. On the automation of computer network simulators. In *Proceedings of the International Conference on Simulation Tools and Techniques (SIMUTools)*, pages 49:1–49:10, 2009.

[148] L. F. Perrone, C. S. Main, and B. C. Ward. SAFE: Simulation Automation Framework for Experiments. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 1–12, 2012.

[149] A. Phillips. Examples in SPiM, accessed 5/2013. http://research.microsoft.com/en-us/projects/spim/examples.pdf.

[150] R. L. Plackett and J. P. Burman. The design of optimum multifactorial experiments. *Biometrika*, 33:305–25, 1946.

[151] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[152] J. R. Quinlan. *C4.5: programs for machine mearning.* Morgan Kaufmann, 1st edition, 1992.

[153] R. L. Rardin. *Optimization in operations research.* Prentice Hall, 1997.

[154] M. Rathinam, L. R. Petzold, Y. Cao, and D. T. Gillespie. Stiffness in stochastic chemically reacting systems: the implicit tau-leaping method. *Journal of Chemical Physics*, 119:12784–12794, 2003.

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

[156] A. Rojnuckarin, D. R. Livesay, and S. Subramaniam. Bimolecular reaction simulation using weighted ensemble brownian dynamics and the University of Houston brownian dynamics program. *Journal of Biophysics*, 79(2):686–693, 2000.

[157] L. Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1–2):1–39, 2010.

[158] S. J. Russel and P. Norvig. *Artificial intelligence: a modern approach.* Prentice Hall, 2nd edition, 2003.

[159] S. Rybacki, J. Himmelspach, F. Haack, and A. M. Uhrmacher. WorMS — a framework to support workflows in m&s. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 716 – 727, 2011.

[160] S. Rybacki, J. Himmelspach, E. Seib, and A. M. Uhrmacher. Using workflows in modeling and simulation software. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 535–545, 2010.

[161] S. Rybacki, S. Leye, J. Himmelspach, and A. M. Uhrmacher. Template and frame based experiment workflows in modeling and simulation software with WORMS. In *Proceedings of the International Workshop on Scientific and Engineering Workflows: Advances in Data and Event-Driven Workflows (SWF)*, pages 25–32, 2012.

[162] R. G. Sargent. Verification, validation, and accreditation of simulation models. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 50–59, 2000.

[163] R. G. Sargent. Verification and validation of simulation models. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 157–169, 2008.

[164] Bernd Schmidt. *The art of modelling and simulation.* SCS Publishing House, 2001.

[165] K. Schmidt. LoLA a Low Level Analyser. In *Proceedings of the International Conference on Application and Theory of Petri Nets (ICATPN)*, volume 1825, pages 465–474, 2000.

[166] L. W. Schruben. Detecting initialization bias in simulation output. *Operations Research*, 30(3):151–153, 1982.

[167] R. C. Seacord, D. Mundie, and S. Boonsiri. K-BACEE: knowledge-based automated component ensemble evaluation. In *Euromicro Conference*, pages 56–62, 2001.

[168] W. T. Shaw. Sampling student's t distribution — use of the inverse cumulative distribution function. *Journal of Computational Finance*, 9(4):37–73, 2006.

[169] K. A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, 41(1):1–25, 2008.

[170] H. Spencer. *The study of sociology.* D. Appleton, 1896.

[171] S. C. Spratt. An evaluation of contemporary heuristics for the startup problem. Master thesis, University of Virginia, 1998.

[172] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction.* MIT Press, 1998.

[173] J. R. Taylor. *An introduction to error analysis.* University Science Books, 2nd edition, 1987.

[174] A. Tockhorn, C. Cornelius, H. Saemrow, and D. Timmermann. Modeling temperature distribution in Networks-on-Chip using RC-circuits. In *Proceedings of the International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pages 229–232, april 2010.

[175] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 2, pages 230–265, 1936.

[176] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem: A correction. In *Proceedings of the London Mathematical Society*, volume 2, pages 544–546, 1938.

[177] UC Berkley EECS Dept. Ptolemy project - Ptolemy II, accessed 5/2013. http://ptolemy.berkeley.edu/ptolemyII/.

[178] A. Unger and H. Schumann. Visual support for the understanding of simulation processes. In *Proceedings of the Visualization Symposium*, pages 5–64, 2009.

[179] W. M. P. van der Aalst. Structural characterizations of sound workflow nets. Technical report, Eindhoven University of Technology, 1996.

[180] W. M. P. van der Aalst. Three good reasons for using a petri-net-based workflow management system, 1996.

[181] W. M. P. van der Aalst and K. M. van Hee. *Workflow Management: Models, Methods, and Systems*, volume 1 of *MIT Press Books.* The MIT Press, 2004.

[182] W. M. P. van der Aalst, K. M. van Hee, and G. J. Houben. Modelling workflow management systems with high-level petri nets. *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms)*, pages 31–50, 1994.

[183] E. Verbeek and W. M. P. Van Der Aalst. Woflan 2.0: a Petri net-based workflow diagnosis tool. In *Proceedings of the International Conference on Application and Theory of Petri Nets (ICATPN)*, pages 475–484, 2000.

[184] R. Vilalta and Y. Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18:77–95, 2002.

[185] R. Vitali, A. Pellegrini, and F. Quaglia. Autonomic Log/Restore for advanced optimistic simulation systems. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 319–327, 2010.

[186] D. Waltemath, R. Adams, D. Beard, F. T.Bergmann, U. S. Bhalla, R. Britten, V. Chelliah, M. T. Cooling, J. Cooper, E. J. Crampin A. Garny, S. Hoops, M. Hucka, P. Hunter, E. Klipp, C. Laibe, A. K. Miller, I. Moraru, D. Nickerson, P. Nielsen, M. Nikolski, S. Sahle, H. M. Sauro, H. Schmidt, J. L. Snoep, D. Tolle, O. Wolkenhauer, and N. Le NovÃ¨re. Minimum Information About a Simulation Experiment (MIASE). *PLoS Computational Biology*, 7(4):e1001122, 04 2011.

[187] T. Wegner, C. Cornelius, M. Gag, A. Tockhorn, and A. M. Uhrmacher. Simulation of thermal behavior for networks-on-chip. In *Proceedings of the NORCHIP Conference*, pages 1–4, 2010.

[188] T. Wegner, M. Gag, and D. Timmermann. Impact of proactive temperature management on performance of Networks-on-Chip. In *Proceedings of the International Symposium on System on Chip (SoC)*, pages 116–121, 2011.

[189] S. A. White. Workflow patterns with BPMN and UML. Technical report, IBM, 2004.

[190] K. P. White Jr. An effective truncation heuristic for bias reduction in simulation output. *Simulation*, 69(6):323–334, 1997.

[191] K. P. White Jr., M. J. Cobb, and S. C. Spratt. A comparison of five steady-state truncation heuristics for simulation. In *Proceedings of the Winter Simulation Conference*, pages 755–760, 2000.

[192] J. R. Wilson and A. A. B. Pritsker. Evaluation of startup policies in simulation experiments. *Simulation*, 31:79–89, 1978.

[193] J. R. Wilson and A. A. B. Pritsker. A survey of research on the simulation startup problem. *Simulation*, 31(2):55–58, 1978.

[194] I. H. Witten and F. Eibe. *Data mining: practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, 1999.

[195] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[196] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[197] L. Petzold Y. Cao, H. Li. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *Journal of Chemical Physics*, 121:4059–4067, 2004.

[198] P. Yang, Y. Hwa Yang, Bing B. B. Zhou, and A. Y. Zomaya. A review of ensemble methods in bioinformatics. *Current Bioinformatics*, 5(4), 2010.

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

List of Figures

List of Tables

List of Symbols

General Symbols

c	confidence interval
σ_X	standard deviation for sample X
e	error tolerance
min	minimum of a set of numerical values
$\mathbb{M}$	model space
$\mathbb{I}_M$	set of possible input parameter settings for model M
$\mathcal{I}_M$	set of possible input parameters for model M
I	an input parameter setting
$\mathbb{O}_M$	set of possible outputs (objectives) of model M
O	a set of outputs
Z_c	c-quantil of the standard normal distribution
$Dist_c$	c-quantil of a given distribution $Dist$
X	a stochastic sample
SA	sample space
ST	set of possible values of a statistic
D_B	detector for the end of the initial bias of a time series
SSE	steady state (mean) estimator
$\perp$	*null* element for steady state estimation
S	a stochastic process
$\widetilde{S}$	steady state for process S
$\hat{S}$	estimated steady state mean for process S
d	relative deviation between real steady state and estimated steady state mean
sr	success rate of a steady state estimator
$\vec{y}$	a time series

Symbols for the NAM Method Experiment

k	rate of diffusional association
D	diffusion coefficient
k_D	rate for two particles of reaching a certain separation
B^∞	probability that the moving particle will have at least one collision with the fixed particle rather than escape into infinity
B	probability of two particles to react
B_{calc}	analytical solution for B
B_{sim}	simulated estimate for B
a	reaction radius of the fixed particle
b	b-surface (radius where the moving particle is placed)
q	q-surface (exit radius of the moving particle)

Symbols for the Wnt/β-catenin Signaling Pathway Experiment

$Axin$	Axin
$AxinP$	phosphorylated Axin
βcyt	β-catenin in the cytosol
βnuc	β-catenin in the nucleus
$k_{\mathrm{Ap}\Rightarrow\mathrm{A}}$	dephosphorylation rate of$AxinP$, that is inhibited by Wnt
$k_{\mathrm{A}\rightarrow\mathrm{Ap}}$	constant phosphorylation rate of $Axin$
$k_{\mathrm{Ap}\downarrow}$	constant decay rate of $AxinP$
$k_{\mathrm{A}\downarrow}$	constant decay rate of $Axin$
$k_{\beta\Downarrow}$	decay rate of βcyt, that is inhibited by $AxinP$
$k_{\beta\uparrow}$	constant production rate of βcyt
$k_{\beta\downarrow}$	constant decay rate of βcyt
$k_{\mathrm{W}\downarrow}$	constant decay rate of Wnt
$k_{\beta\mathrm{in}}$	movement rate of β-catenin from cytosol to nucleus
$k_{\beta\mathrm{out}}$	movement rate of β-catenin from nucleus to cytosol
$k_{\mathrm{Ap}\rightarrow\mathrm{A}}$	constant dephosphorylation rate of$AxinP$
$k_{\mathrm{A}\uparrow}$	constant production rate of $Axin$
$\mathcal{S}$	function for calculating the sensitivity of a set of input parameters
$\mathcal{S}_P$	function for calculating the sensitivity of a single input parameter
$\mathcal{D}$	distance between results of base configuration and currently executed configuration
arg_p	value of parameter p of a given parameter configuration
$\mathcal{R}$	simulation results for a model executed with a given parameter configuration
δ_{eq}	function for calculating the euclidean distance between two time series

Symbols for the NoC Experiment

T_i	threshold for component's temperature, at which the TMU starts to intervene
T_a	threshold for component's activity, at which the TMU starts to intervene
B_u	upper bound for the IPC's temperature to allow task rescheduling
B_l	lower bound for the IPC's temperature to allow task rescheduling
$BLOCK$	RC circuit resolution, where each component is represented by one RC tile
$RES1$	RC circuit resolution, where routers are represented by one RC tile
$RES2$	RC circuit resolution, where routers are represented by 2×2 RC tiles
$RES3$	RC circuit resolution, where routers are represented by 2×2 RC tiles

Symbols for Algorithm Selection & Composition

$\mathbb{P}$	problem space
$\mathbb{P}_{it}$	set of iterated problems
A	set of available algorithms
$\mathbb{A}$	algorithm space
$\mathbb{F}$	feature set
F	feature extractor
$\mathbb{F}_P$	problem feature set
F_P	problem feature extractor
$\mathcal{F}_P$	set of problem feature extractors
$\mathbb{F}_S$	state feature set
F_S	state feature extractor
$\mathcal{F}_S$	set of state feature extractors
H_F	feature history of an SPS
$\mathbb{H}_F$	set of feature histories
P	performance mapping
S	selection mapping
PS	problem solver
RES	set of possible results a problem solver PS can produce
REQ	set of possible requests a problem solver PS can formulate
ANS	set of possible answers a problem solver PS can accept
η	answer function for the requests of a problem solver PS
H_R	request history of a problem solver PS
$\mathbb{H}$	set of request histories
$\mathbb{S}$	set of problem solver states
$\mathbb{S}_{synth}$	set of SPS states
AS	set of algorithm state information
SPS	synthetic problem solver
ς	selection function of an SPS
κ	composition function of an SPS
TD	training data set
G	a problem generator

List of Acronyms

AMS	Analog Mixed Signal
AOTA	Adaptive Online Time Allocation
ASP	Algorithm Selection Problem
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
DBMS	DataBase Management System
DOE	Design Of Experiments
EDL	Experiment Description Language
GUI	Graphical User Interface
HCI	Human Computer Interaction
ICN	Information Control Net
IPC	Intellectual Property Core
JDBC	Java DataBase Conectivity
LHC	Large Hadron Collider
M&S	Modeling and Simulation
MIASE	Minimum Information About a Simulation Experiment
MRIP	Multiple Run In Parallel
MSE	Mean Squared Error
NAM	Northrup-Allison-McCammon
NoC	Network on Chip
RC	Resistor-Capacitor
RISE	Restful Interoperability Simulation Environment
RNG	Random Number Generator
SED-ML	Simulation Experiment Description Markup Language
SESSL	Simulation Experiment Specification via a Scala Layer
SSA	Stochastic Simulation Algorithm
SSE	Steady State Estimator

TMU	Thermal Management Unit
URI	Uniform Resource Identifier
V&V	Verification and Validation
XML	eXtensible Markup Language
XPDL	XML Process Definition Language

Thesis Statements

Title: Toward Guiding Simulation Experiments
Name: Stefan Leye

1.a Executing a simulation experiment is a complex task, which can overwhelm non-expert users. Hence, guidance through the experiment process is required to reduce the amount of decisions, such users have to make.

1.b A reduction of decisions is possible by structuring the experiment process and identifying typical tasks that have to be considered. Algorithm selection & composition techniques can support users in applying the right methods for such tasks.

1.c The simulation experiment process can be structured according to the six tasks: specification, configuration, model execution, data collection, analysis, and evaluation. Thereby, the experiment gets a clear and explicit structure, complying with different types of simulation experiments and making it more transparent to users.

1.d A simulation experiment tool should incorporate those tasks explicitly, and allow a flexible integration of arbitrary experiment methods.

Realizing the Experiment Process

2.a A flexible implementation of tasks is possible by extending the experimentation layer of the simulation framework JAMES II and exploiting its plugin-based design. Tasks can be realized as plugin types, while concrete methods are integrated as plugins and through clear interfaces.

2.b The extended experimentation layer allows the execution of different experiments types from different application domains.

2.c The experimental structure can be mapped to workflows to profit from the documentation features that ship with workflow management systems like WorMS. Thereby, the notion of templates and frames allows a realization of tasks as templates and concrete functionality as frames that are used to fill such templates.

Algorithm Selection and Composition

3.a A general interface can be identified, that maps different algorithms applied in simulation experiments and incorporates iterative problem solving.

3.b Algorithms following the interface can be combined in a synthetic problem solver which considers problem and algorithm features for composing them to an advanced version.

3.c Synthetic problem solvers allow a mapping of various composition schemes, like algorithm selection, algorithm portfolios, algorithm ensembles, and online adaptation.

3.d Synthetic problem solvers can be made accessible in an extension of the algorithm selection framework SASF.

3.e A synthetic problem solver instance that composes steady state estimators by considering time series features and estimator results, can be more robust and accurate than its base-line estimators.

3.f It is possible to generate synthetic problem solvers instances for statistical analysis that are optimized either for input sample sizes or the number of analysis iterations.

Resume

Personal Data

Name	Stefan Leye
Date of Birth	16-Decembre-1982
Place of Birth	Neubrandenburg
Nationality	Germany

Education

since 5/2008	research fellow at University of Rostock Rostock, Germany
since 5/2008	PhD student at University of Rostock Rostock, Germany
4/2008	university Diploma (Diplom) from University of Rostock Rostock, Germany
10/2003 - 4/2008	undergrate student of computer science at University of Rostock Rostock, Germany
10/2006 - 3/2007	research visit at The Microsoft Research - University of Trento Centre for Computational and Systems Biology, Trento, Italy
6/1993 - 6/2002	high school diploma (Abitur) from Curie-Gymnasium Neubrandenburg Neubrandenburg, Germany

Erklärung

Ich, Stefan Leye, erkläre, dass ich die vorliegende Dissertationsschrift mit dem Thema: “Toward Guiding Simulation Experiments” selbständig, ohne die (unzulässige) Hilfe Dritter und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, 25. Oktober 2013

Ort, Datum

Unterschrift