# Simulator Adaptation at Runtime for Component-Based Simulation Software

Dissertation

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

vorgelegt von

Tobias Helms, geb. am 01.08.1988 in Schwerin

Rostock, 03.05.2017

urn:nbn:de:gbv:28-diss2017-0134-6

**Principal Advisor:**

Prof. Dr. Adelinde M. Uhrmacher (Universität Rostock, Germany)

**External Reviewer:**

1. Prof. Dr. Francesco Quaglia (Sapienza Università di Roma, Italy)

2. Prof. Dr. Georgios K. Theodoropoulos, (Durham University, England and Southern University of Science and Technology, China)

**Date of Submission:** 03.05.2017

**Date of Defense:** 25.07.2017

# Abstract

Component-based simulation software can provide many opportunities to compose and configure simulators, resulting in an algorithm selection problem for the user of this software. Further, as the state and structure of a model may vary during a simulation run, the computational demands might also change during runtime. Therefore, it is not only necessary to select a suitable simulator for executing a simulation run, but this selection must regularly be reconsidered to adapt the chosen simulator to changed computational demands. While this is a general and cross-cutting concern, most adaptation schemes for simulators are tailored to specific application scenarios that cannot be reused straightforwardly for other scenarios. Therefore, this thesis aims to automate the selection and adaptation of simulators at runtime in an application-independent manner. Further, it explores the potential of tailored and approximate simulators — in this thesis concretely developed for the modeling language ML-Rules— supporting the effectiveness of the adaptation scheme.

Specifically, for the automatic selection and adaptation of simulators at runtime, a flexible and generic adaptive simulator is developed and integrated into the modeling and simulation framework JAMES II. The adaptive simulator encapsulates available simulators applicable to a specific problem and employs reinforcement learning to explore and exploit the performance of these simulators. As it uses the encapsulated simulators to calculate the state transitions of a model, it is not restricted to any modeling language, but it can be applied to all modeling approaches available in JAMES II. To improve the learning efficiency of the adaptive simulator, state space generalization methods are applied. Further, different techniques to trigger adaptations are explored, e.g., a changepoint detection method monitoring the event throughput is integrated into the adaptive simulator.

A pool of efficient simulators is a prerequisite for the effectiveness of the adaptive simulator. Therefore, in addition to the adaptive simulator itself, in this thesis tailored and approximate simulators are developed and explored concretely for the modeling language ML-Rules. Due to its expressiveness, it poses various computational challenges tackled by the developed simulators. The efficiency of these simulators is illustrated with complex ML-Rules models used in simulation studies.

# Zusammenfassung

Komponenten-basierte Simulationssoftware kann viele Möglichkeiten zur Komposition und Konfiguration von Simulatoren bieten und damit zu einem Konfigurationsproblem für Nutzer dieser Software führen. Ausgelöst durch Zustands- und Strukturänderungen eines Modells können sich die rechentechnischen Anforderungen an den Simulator außerdem zur Laufzeit verändern. Daher ist es nicht ausreichend einen geeigneten Simulator zur Ausführung eines Simulationslaufs einmalig auszuwählen — die Auswahl muss regelmäßig überprüft und gegebenenfalls an veränderte Anforderungen angepasst werden. Obwohl dies ein genereller Aspekt ist, sind die meisten Adaptionsmechanismen für Simulatoren auf bestimmte Anwendungsszenarien zugeschnitten. Das Ziel dieser Arbeit ist daher die Entwicklung einer generischen und automatisierten Auswahl- und Adaptionsmethode für Simulatoren. Darüber hinaus wird das Potential von spezifischen und approximativen Simulatoren anhand der Modellierungssprache ML-Rules untersucht, welche die Effektivität des entwickelten Adaptionsmechanismus erhöhen können.

Zur automatischen Selektion und Adaption von Simulatoren zur Laufzeit wird ein flexibler und generischer adaptiver Simulator entwickelt und in das Modellierungs- und Simulationsframework JAMES II integriert. Der adaptive Simulator kapselt vorhandene Simulatoren und verwendet verstärkendes Lernen um die Leistung der Simulatoren zu untersuchen und auszunutzen. Da die gekapselten Simulatoren zur Berechnung der eigentlichen Simulationsläufe verwendet werden, ist der adaptive Simulator nicht auf eine Modellierungsspache beschränkt, sondern kann für jede Modellierungssprache, die in JAMES II verfügbar ist, verwendet werden. Zur Verbesserung der Lerneffizienz des adaptiven Simulators werden Generalisierungsmethoden für Zustandsräume angewendet und evaluiert. Zusätzlich werden verschiedene Techniken zum Auslösen einer Adaption untersucht, beispielsweise eine Methode zur Erkennung von deutlichen Änderungen der Ereignisrate.

Eine Menge von effizienten Simulatoren ist eine Grundvoraussetzung für die Effektivität des entwickelten adaptiven Simulators. Daher werden in dieser Arbeit, neben dem adaptiven Simulator selbst, zusätzlich spezifische und approximative Simulatoren für die Modellierungssprache ML-Rules entwickelt. Durch dessen Ausdrucksstärke entstehen verschiedene rechentechnische Herausforderungen, zu deren Bewältigung die entwickelten Simulatoren beitragen. Die Effizienz dieser Simulatoren wird anhand verschiedener komplexer ML-Rules Modelle, welche in Simulationsstudien verwendet wurden, illustriert.

# Acknowledgments

Writing a thesis is a long, challenging and fascinating journey including demanding as well as exhausting phases. I would not have been able to finish this work without so many people helping me. I express my greatest gratitude to Lin for always supporting me during this journey with her motivation and knowledge and for giving me the time to focus on this thesis. I also thank Professor Francesco Quaglia and Professor Georgios Theodoropoulos for reviewing this thesis.

Special thanks go to Roland for arousing my interest to modeling, simulation and algorithm selection topics and for always being fascinated by so many open problems. I also want to thank in particular Danhua, John and Tom for being valuable and constructive colleagues and friends. I had a great time at the modeling and simulation group in general and also want to thank all former and all current colleagues.

During the last years, I had the honor to supervise many students writing their Bachelor's or Master's theses contributing to my work. Thank you Marcel, Steffen, Oliver, Sebastian, Jakob, and Pia for always being motivated and ambitious while working on your theses.

I am also deeply grateful to my friends and family for all the support, understanding and fun over the last years. In particular, I deeply thank Mareen and Paulina for being the most important individuals in my life and making it wonderful.

# Contents

**List of Figures** **x**

**List of Algorithms** **xiii**

**1 Introduction** **1**
1.1 Contribution . . . . . 3
1.2 Outline . . . . . 4
1.3 Bibliographic Note . . . . . 4

**2 Adaptivity in Modeling & Simulation** **7**
2.1 Simulation Experiments . . . . . 8
2.2 Continuous Simulation . . . . . 12
2.3 Parallel Discrete Event Simulation . . . . . 15
2.4 Simulation of Biochemical Reaction Networks . . . . . 18
2.4.1 $\tau$-leaping . . . . . 20
2.4.2 Combining Discrete and Continuous Simulators . . . . . 22
2.5 Simulator Selection for Discrete Event Simulation . . . . . 25
2.5.1 Simulators as Selection Trees . . . . . 25
2.5.2 Simulator Selection via Supervised Learning . . . . . 27
2.5.3 Simulator Selection via Unsupervised Learning . . . . . 28
2.6 Summary . . . . . 30

**3 Adaptive Software** **34**
3.1 Concepts for Compositional Adaptations . . . . . 39
3.2 Techniques to Implement Adaptive Software . . . . . 42
3.3 Summary . . . . . 44

**4 The Adaptive Simulator — Compositional Simulator Adaptation at Runtime** **45**
4.1 Requirements . . . . . 46

4.2 The Structure of the Adaptive Simulator . . . 47
4.3 State Space Generalization . . . 55
4.3.1 Decision Boundary Partitioning . . . 60
4.3.2 Adaptive Vector Quantization . . . 64
4.4 Adaptation Conditions . . . 72
4.4.1 Changepoint Detection for Adaptive Simulation Algorithms . 74
4.5 Implementation & Integration in JAMES II . . . 79
4.5.1 Information Retrieval . . . 81
4.5.2 Adaptation Condition . . . 84
4.5.3 Value Function . . . 85
4.6 Measuring Adaptation Performance . . . 87
4.7 Limitations and Open Challenges . . . 88
4.7.1 Testing Component-based Stochastic Simulators . . . 89
4.8 Summary . . . 92

**5 Performance Experiments with the Adaptive Simulator 96**
5.1 Experiments with ML-Rules . . . 97
5.1.1 Introduction . . . 98
5.1.1.1 Enzyme-Substrate-Product Model . . . 98
5.1.1.2 Attributed Species . . . 99
5.1.1.3 Compartments . . . 102
5.1.1.4 Multi-Level Rules . . . 106
5.1.1.5 Functions on Solutions . . . 107
5.1.2 Experiments . . . 108
5.1.2.1 Experiments with a Benchmark Model . . . 109
5.1.2.2 Experiments with Complex Models . . . 115
5.1.2.3 Changepoint Detection Experiment . . . 116
5.1.2.4 Dynamic State Space Generalization Experiment . . 118
5.2 Other Modeling Formalisms . . . 120
5.2.1 Species-Reactions (SR) . . . 120
5.2.2 PDEVS . . . 121
5.3 Summary . . . 125

**6 Tailored and Approximate Simulators - A Case Study with ML-Rules 127**
6.1 Tailored Simulators for ML-Rules . . . 130
6.1.1 Static Species and Reaction Sets . . . 131
6.1.1.1 Results with the Wnt/$\beta$-catenin Pathway Model . . 132
6.1.2 Species Bindings . . . 133

6.1.2.1 Reactant Swapping . . . . . . . . . . . . . . . . . . . . 134
6.1.2.2 Results with a Mitochondria Model . . . . . . . . . . 136
6.2 $\tau$-leaping for ML-Rules . . . . . . . . . . . . . . . . . . . . . . . . . 138
6.2.1 Results and Accuracy Analysis with Visual Analytics . . . . . 142
6.3 Hybrid Simulator for ML-Rules . . . . . . . . . . . . . . . . . . . . . 144
6.3.1 Reaction Partitioning . . . . . . . . . . . . . . . . . . . . . . . 145
6.3.2 Calculation of Deterministic Reactions . . . . . . . . . . . . . 146
6.3.3 Results with a Benchmark Model . . . . . . . . . . . . . . . . 147
6.3.4 Results with a Dictyostelium Discoideum Model . . . . . . . . 149
6.3.5 Parallel Execution of Stochastic Reactions . . . . . . . . . . . 152
6.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152

**7 Conclusions and Outlook 156**
7.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156
7.2 Outlook . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161

**Bibliography 163**

**A ML-Rules Models 182**
A.1 Cell Cycle Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182
A.2 Endocytosis Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
A.3 Wnt/$\beta$-catenin Model . . . . . . . . . . . . . . . . . . . . . . . . . 185
A.4 Simplified Lipid Raft Model . . . . . . . . . . . . . . . . . . . . . . 187
A.5 Dictyostelium Discoideum Model . . . . . . . . . . . . . . . . . . . 188

# List of Figures


2.1 Six tasks of a simulation experiment [116]. . . . . . . . . . . . . . . . . 9
2.2 Runtime influence of observation [77]. . . . . . . . . . . . . . . . . . . . . 11
2.3 The algorithm selection problem as defined by Rice [166]. . . . . . . . . 12
2.4 Equations of the Domarnd Prince 54 method [39]. . . . . . . . . . . . . 14
2.5 Possible improvement of reaction selection . . . . . . . . . . . . . . . . . 20
2.6 Main packages of JAMES II [87]. . . . . . . . . . . . . . . . . . . . . . . . 26
2.7 Example of a paramter block . . . . . . . . . . . . . . . . . . . . . . . . . 27

3.1 Spectrum of adaptivity . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.2 Original and extended classification of adaptivity based on [135] . . . 37
3.3 The MAPE-K control loop based on [105] . . . . . . . . . . . . . . . . . 41
3.4 UML class diagram of the wrapper pattern [60] . . . . . . . . . . . . . 43

4.1 The wrapper pattern and `Adaptive Simulator` . . . . . . . . . . . . . 48
4.2 The basic reinforcement learning model [185] . . . . . . . . . . . . . . . 49
4.3 The main components of the `Adaptive Simulator` . . . . . . . . . . 52
4.4 State handling of the `Adaptive Simulator` . . . . . . . . . . . . . . . . 53
4.5 ML-Rules benchmark model runtimes [76]. . . . . . . . . . . . . . . . . 57
4.6 Runtime Boxplot of `Adaptive Simulator` for ML-Rules benchmark model and grid generalization. . . . . . . . . . . . . . . . . . . . . . . . 58
4.7 Grid-based state space generalizations . . . . . . . . . . . . . . . . . . . 59
4.8 State splitting by the DBPA. . . . . . . . . . . . . . . . . . . . . . . . . 60
4.9 State space extension and the DBPA . . . . . . . . . . . . . . . . . . . 62
4.10 State space generalizations created by the `Adaptive Simulator` using the DBPA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.11 Runtime of `Adaptive Simulator` with DBPA compared to grid-based generalizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
4.12 Codebook extension by the AVQ. . . . . . . . . . . . . . . . . . . . . . 66
4.13 Codeword merge by the AVQ. . . . . . . . . . . . . . . . . . . . . . . . 67
4.14 State space generalizations of the `Adaptive Simulator` using the AVQ 71

4.15 Runtime of `Adaptive Simulator` using AVQ compared to DBPA and grid-based generalization. . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.16 Run length illustration [1]. . . . . . . . . . . . . . . . . . . . . . . . . 75
4.17 Overview class diagram of the `Adaptive Simulator`. . . . . . . . . . 82
4.18 JAMES II context illustration. . . . . . . . . . . . . . . . . . . . . . . 83
4.19 Class diagram adaptation condition plugin type. . . . . . . . . . . . . 84
4.20 Class diagram value function plugin type. . . . . . . . . . . . . . . . . 86
4.21 Observation strategy to test simulators. . . . . . . . . . . . . . . . . . 90
4.22 Trajectory distribution comparison. . . . . . . . . . . . . . . . . . . . 91

5.1 ML-Rules enzyme-substrat-product model [82] . . . . . . . . . . . . . 99
5.2 ML-Rules cell cycle model [82] . . . . . . . . . . . . . . . . . . . . . . 101
5.3 Illustration of reaction execution with population-based compartments 103
5.4 Mapping of old and new contexts . . . . . . . . . . . . . . . . . . . . 105
5.5 ML-Rules shuttling model [82] . . . . . . . . . . . . . . . . . . . . . . 106
5.6 An abstract endocytosis model illustrating the creation and fusion of compartments. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.7 The benchmark model from [75] written in the current ML-Rules syntax. 110
5.8 ML-Rules benchmark model runtimes [76]. . . . . . . . . . . . . . . . 110
5.9 Detailed runtime results of the ML-Rules benchmark model [76]. . . . 111
5.10 Dynamic regret with different state space resolutions [76]. . . . . . . . 112
5.11 Dynamic regret for several policies and the ML-Rules benchmark model [76]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
5.12 Runtime results for the complex models [76]. . . . . . . . . . . . . . . 115
5.13 Number of executed adaptations [80]. . . . . . . . . . . . . . . . . . . 117
5.14 Performance results for the Wnt/$\beta$-catenin pathway model and the Bayesian changepoint detection algorithm [80]. . . . . . . . . . . . . . 117
5.15 Dynamic regret for the ML-Rules benchmark model and different state space generalizations. . . . . . . . . . . . . . . . . . . . . . . . . . . 119
5.16 Performance results of dynamic state space representations in ML-Rules [79]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
5.17 PDEVS Smoke Detector Example . . . . . . . . . . . . . . . . . . . . 121
5.18 Dynamic regret of the `Adaptive Simulator` executing the forest fire model in PDEVS [80]. . . . . . . . . . . . . . . . . . . . . . . . . . 124

6.1 Automatic validity check for simulators . . . . . . . . . . . . . . . . . 129
6.2 Wnt/$\beta$-catenin pathway model ML-Rules results. . . . . . . . . . . . 133
6.3 ML-Rules model with bindings . . . . . . . . . . . . . . . . . . . . . . 134
6.4 Runtime results of the tailored ML-Rules simulator for bound species [79]. 137

6.5 Impact of $\mu$. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
6.6 Firing probabilities based on Poisson distribution. . . . . . . . . . . . 140
6.7 Firing numbers based on two independent Poisson distributions . . . 141
6.8 Visual Analytics Tool presented in [123] . . . . . . . . . . . . . . . . . 142
6.9 Accuracy illustrations [78]. . . . . . . . . . . . . . . . . . . . . . . . . 143
6.10 Steady State Analysis and Stationary Distributions . . . . . . . . . . 147
6.11 ML-Rules benchmark for hybrid simulators . . . . . . . . . . . . . . . 149
6.12 Hybrid ML-Rules simulator results. . . . . . . . . . . . . . . . . . . . 150
6.13 Hybrid ML-Rules simulator results for the dictyostelium discoideum amoebas model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 151

# List of Algorithms


2.1 The Direct Method [63]. . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.2 Reaction selection of the SSA [63]. . . . . . . . . . . . . . . . . . . . . 19
2.3 Hybrid simulator for biochemical reaction networks from [71]. . . . . 23
4.1 Pseudo-code of the adaptive simulator. . . . . . . . . . . . . . . . . . . 51
4.2 Pseudo-code of the AVQ algorithm [114]. . . . . . . . . . . . . . . . . . 65
4.3 Pseudo-code of our intial AVQ algorithm. . . . . . . . . . . . . . . . . . 68
4.4 Pseudo-code of our AVQ algorithm (without merge). . . . . . . . . . 69
4.5 Merging part of our AVQ algorithm. . . . . . . . . . . . . . . . . . . . 70
4.6 Pseudo-code of our adaptation condition method. . . . . . . . . . . . 77
6.1 Java-code of reactant swapping. . . . . . . . . . . . . . . . . . . . . . . 136
6.2 Threshold calculation for reaction partitioning. . . . . . . . . . . . . . 145
6.3 Hybrid Simulator for ML-Rules . . . . . . . . . . . . . . . . . . . . . . 148
6.4 Hybrid Simulator for ML-Rules with multiple stochastic events per step. 153

# Chapter 1

# Introduction

> A good algorithm is the most important thing when it comes to fast performance.
>
> Scott Oaks [145]

In 2001, IBM published a perspective paper about autonomic computing [92]. The fundamental message of this paper referring to software systems is that "the obstacle is complexity. Dealing with it is the single most important challenge facing the I/T industry. It is our next *Grand Challenge*". This challenge has also reached the modeling and simulation community. Models are getting more complex, e.g., [103, 141], simulation studies are getting more complex [100, 152], and simulation systems are getting more complex [61, 197, 84].

Modeling and simulation are established tools to study existing or theoretical systems. Their usage is motivated by various reasons, e.g., it could be too expensive or morally not acceptable to study the system of interest directly. Furthermore, any aspect of a model can be controlled, i.e., input and model parameters can be changed as needed and all model properties are observable. Therefore, building simulation models and performing experiments with them are valuable methods to gain information about systems of interest.

To keep control about complexity, domain specific modeling languages often enable modeler to create compact and succinct models that are easier to understand and less error-prone than models written in a general-purpose programming language [192]. Moreover, composing and fusing models are established methods to built complex models based on existing ones [153]. For simulation studies, domain specific languages like the simulation experiment specification SESSL can also help to keep control about complexity [53]. Besides, workflow concepts might be applied [167] and guidance support to execute simulation experiments can be used [116, 154].

The complexity of simulation systems can be handled by using essential concepts of software engineering, e.g., abstraction, separation of concerns, reuse, and design patterns. These concepts have been the basis for the development of the modeling and simulation framework JAMES II [87]. The framework realizes a component-based structure and separates basic parts referring to modeling and simulation, e.g., it requires a strict separation between a model and a simulation algorithm. It serves as a basis to develop concrete simulation systems by reusing its architecture, many features and processes. The component-based structure allows developing different implementations for each component and to create compositions on the fly during runtime. Nevertheless, this flexibility comes with its own challenge: selection decisions have to be made. Which simulation algorithm shall be used and how shall it be configured? Which steady state estimator shall be used? Which event queue fits the requirements best?

Like suggested in IBM's perspective paper, automatic approaches and paradigms like self-adaptive software [105] or programming by optimization [91] are needed to relieve the user from configuring and maintaining such a complex software system that offers a vast variety of compositional options. Further, automatic concepts also relieve the developers from premature commitments, i.e., selecting concrete components, algorithms and data structures for tasks whose requirements are either not fully clear during the development or whose requirements depend on concrete application scenarios.

In general, simulation experiments do not only include the execution of one simulation run, but they are complex tasks involving diverse phases and methods to be executed [116]. For example, model configurations have to be selected purposefully using statistical methods like Latin hypercube sampling [124]. Further, suitable replication criteria and simulation termination criteria have to be determined [104, p. 522ff.]. The observation of model states is another challenging task [77]. All these tasks influence the performance of a simulation experiment and for all of them, selection decisions have to be made.

However, automatic selection and adaptation methods are often developed explicitly for simulation algorithms, i.e., simulators, to improve the runtime performance of simulation runs, see Chapter 2. Adaptations of simulators are motivated by different "simulation run phases" with different computational requirements [142]. Adapting a simulator during runtime induces complex challenges. For example, suitable adaptation trigger must be identified. Moreover, features must be determined to distinguish the "simulation run phases". Most existing adaptation mechanisms solve these challenges concretely for specific application-scenarios, i.e., the developer of these mechanisms determine suitable adaptation trigger, implement suitable adaptation functions etc.

Although these methods can be effective, they cannot be applied straightforwardly to other application scenarios. In general, developing an adaptation method for simulators in an application-independent manner is complex as it induces further challenges, e.g., learning methods have to be applied enabling the adaptation method to learn how to perform suitable adaptations.

## 1.1 Contribution

This thesis aims to automate the selection and adaptation of simulators at runtime in an application-independent manner. Therefore, initially existing methods adapting simulators are analyzed and categorized based on established features of adaptive algorithms. Based on this analysis, a flexible and generic adaptive simulator (marked as `Adaptive Simulator`) is developed for the automatic selection and adaptation of simulators at runtime. It is integrated into the component-based modeling and simulation framework JAMES II. Thereby, the `Adaptive Simulator` encapsulates available simulators applicable to a specific problem and employs reinforcement learning to explore and exploit the performance of these simulators. By exploiting the component-based architecture of JAMES II, the `Adaptive Simulator` calculates the set of available simulators and all of their configurations automatically. As it uses the encapsulated simulators to calculate the state transitions of a model, it is not restricted to any modeling language, but it can be applied to all modeling approaches available in JAMES II.

To deal with large or infinite state spaces used to distinguish "simulation run phases" [142], we integrate dynamic state space generalization methods into the `Adaptive Simulator`. Furthermore, we apply a changepoint detection method [1] monitoring the event throughput to trigger adaptations.

We illustrate the flexibility of the `Adaptive Simulator` by applying it to three different modeling formalisms: ML-Rules [130], SR [97] and PDEVS [205]. Thereby, we analyze different properties of the `Adaptive Simulator`, e.g., the impact of different multi-armed bandit policies for the action selection on the performance of the `Adaptive Simulator`.

To be efficient, an adaptive mechanism needs a pool of simulators and adaptation options. In this context, expressive modeling languages inducing various computational challenges are suitable candidates to develop different simulators. In the realm of biochemical reaction networks, ML-Rules is such an expressive and complex modeling languages aimed at dynamically nested biochemical reaction networks with attributed entities [130]. Due to its expressiveness, the simulator of ML-Rules offers various points to develop different methods and it is therefore a suitable candidate to explore

automatic adaptation methods. In this thesis, we develop tailored and approximate simulators for ML-Rules achieving significant speed-ups. The efficiency of these simulators is illustrated with complex ML-Rules models used in simulation studies.

## 1.2 Outline

The thesis is organized as follows. Initially, in Chapter 2, opportunities to apply adaptivity in modeling and simulation are illustrated with different application scenarios. In the following Chapter 3, existing categorization approaches for adaptive software in general are mapped to the methods identified in Chapter 2. Key characteristics and approaches to implement complex adaptive software are presented. Chapter 4 presents the concept of the `Adaptive Simulator` performing adaptations during runtime for component-based simulation software. Thereby, methods to deal with large or infinite state spaces and different methods to trigger adaptations are explored. Experiment results with the `Adaptive Simulator` and the modeling languages ML-Rules, SR, and PDEVS are presented in Chapter 5. The potential of tailored and approximate simulators in the context of ML-Rules is shown in Chapter 6. Finally, Chapter 7 concludes the thesis and gives an outlook about future work.

## 1.3 Bibliographic Note

The first version of the `Adaptive Simulator` including experiments with ML-Rules and SR has been published in the following publication.

> Tobias Helms, Roland Ewald, Stefan Rybacki and Adelinde M. Uhrmacher (2013): A Generic Adaptive Simulation Algorithm for Component-based Simulation Systems. Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, pp. 11-22.

A revised version of the `Adaptive Simulator` also including experiments with PDEVS has been published in the following publication. The main difference compared to the initial version of the `Adaptive Simulator` is that an action do not refer to a tuple of a simulator and an adaptation condition, but only simulators represent actions and adaptation conditions are handled separately. Further, we present initial ideas and results for applying state space generalization methods.

> Tobias Helms, Roland Ewald, Stefan Rybacki and Adelinde M. Uhrmacher (2015): Automatic Runtime Adaptation for Component-based Simulation Algorithms.

ACM Transactions on Modeling and Computer Simulation (TOMACS), Volume 26 Issue 1, Article 7, pp 1-24.

We integrated the changepoint detection mechanism to trigger adaptations into the `Adaptive Simulator` in the following publication.

> Tobias Helms, Oliver Reinhardt and Adelinde M. Uhrmacher (2015): Bayesian Changepoint Detection for Generic Adaptive Simulation Algorithms. Proceedings of the 48th Annual Simulation Symposium, pp. 62-69.

State space generalization methods and their integration into the `Adaptive Simulator` are presented in the following publication.

> Tobias Helms, Steffen Mentel, and Adelinde M. Uhrmacher (2016): Dynamic State Space Partitioning for Adaptive Simulation Algorithms. Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools, pp. 149-152.

Referring to ML-Rules, we developed its $\tau$-leaping simulator in the following publication.

> Tobias Helms, Martin Luboschik, Heidrun Schumann and Adelinde M. Uhrmacher (2013): An Approximate Execution of Rule-based Multi-level Models. Proceedings of the 11th International Conference on Computational Methods in Systems Biology, pp. 19-32.

An overview about ML-Rules and the tools we have developed for it including the `Adaptive Simulator` is given in the following publication.

> Tobias Helms, Carsten Maus, Fiete Haack and Adelinde M. Uhrmacher (2014): Multi-level modeling and simulation of cell biological systems with ML-Rules: A Tutorial. Proceedings of the Winter Simulation Conference, pp. 177-191.

The impact of different observation strategies for ML-Rules during a simulation run are analyzed in the following publication.

> Tobias Helms, Jan Himmelspach, Carsten Maus, Oliver Röwer, Johannes Schützel and Adelinde M Uhrmacher (2012): Toward a language for the flexible observation of simulations. Proceedings of the Winter Simulation Conference, pp. 418-430.

We developed a formal semantics for ML-Rules in the following publication. This formal semantics enables us to study the computational challenges of ML-Rules simulations comprehensively and it allows comparing the validity of simulators.

> Tom Warnke, Tobias Helms and Adelinde M. Uhrmacher (2015): Syntax and Semantics of a Multi-Level Modeling Language. Proceedings of the ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, pp. 133-144.

Based on the work with the ML-Rules semantics, we present the potential of tailored and approximate simulators for ML-Rules in the following publication.

> Tobias Helms, Tom Warnke, Carsten Maus and Adelinde M. Uhrmacher (2016): Semantics and Efficient Simulation Algorithms of an Expressive Multi-Level Modeling Language. ACM Transactions on Modeling and Computer Simulation (TOMACS), in press.

# Chapter 2

# Adaptivity in Modeling & Simulation

> I believe the biggest impact on successful software development is motivated, talented developers.
>
> Martin Fowler

The purpose of modeling and simulation is to study existing or theoretical systems. A model is an abstraction of the studied system that should be as complex as necessary and as simple as possible to enable answering questions about the system suitably, see Definition 1. Models can either be physical or mathematical models — the latter "representing a system in terms of logical and quantitative relationships" [104, p.5].

**Definition 1. Model** A model (M) for a system (S) and an experiment (E) is anything to which E can be applied in order to answer questions about (S) [28, p. 5].

There are various reasons to use a model of a system to study it. For example, the system of interest might be too complex to study it practically without abstractions reflected in a model. It could be too expensive or morally not acceptable to study the system directly. Further, any aspect of a model can be controlled, i.e., input and model parameters can be changed as needed and all model properties are observable. If the system of interest does not exist yet, a model must be used to study its behavior.

As written in Definition 1, a model is not only related to a system and the questions that shall be answered by using the model, but it is always also related to an experiment, see Definition 2. Consequently, the validity of a model can only be evaluated by also considering the experiment that shall be performed on the model. Altogether, the terms model and experiment can be used to define the term simulation, see Definition 3.

**Definition 2. Experiment** An experiment is the process of extracting data from a system by exerting it through its inputs [28, p. 4].

**Definition 3. Simulation** A simulation is an experiment performed on a model [110].

A simulation gains data from a model by performing simulation runs, see Definition 4. Algorithms that can execute a simulation run are called simulators, see Definition 5. In case of stochastic models, simulation runs have to be repeated with the same inputs but different seeds for the pseudo random number generators to gain statistically suitable data. Such repeated simulation runs are called replications, see Definition 6. In general, the complexity of a simulation is not restricted. For example, one simulation can include several simulation runs using various input parameter values that are selected based on analyzed intermediate results.

**Definition 4. Simulation Run** A simulation run is a single model execution, i.e., using the initial state of the model with specific input parameter values and calculating successive state transitions until a termination criteria is fulfilled.

**Definition 5. Simulator** A simulator is an algorithm that executes simulation runs.

**Definition 6. Replication** A replication is one simulation run that is repeated several times with the same input parameter values but different seeds for the random number generator.

Following the terminology used by Leye in [116, p.3], in the rest of this thesis, the term **simulation experiment** is used instead of the term **simulation** to emphasize that a simulation typically corresponds to a complex experiment including the execution of a set of simulation runs and further complex tasks, e.g., analyzing simulation run results. Besides, the verb **to simulate** is used to describe the process of performing an individual simulation run.

## 2.1 Simulation Experiments

A simulation experiment is a complex task including various aspects. Leye identified six basic parts of a simulation experiment [116]:

- **Specification:** Ideally, a formal representation of the experiment goal.
- **Configuration of model parameters:** Determine a set of model input parameters chosen e.g., by parameter scan algorithms or parameter search methods.

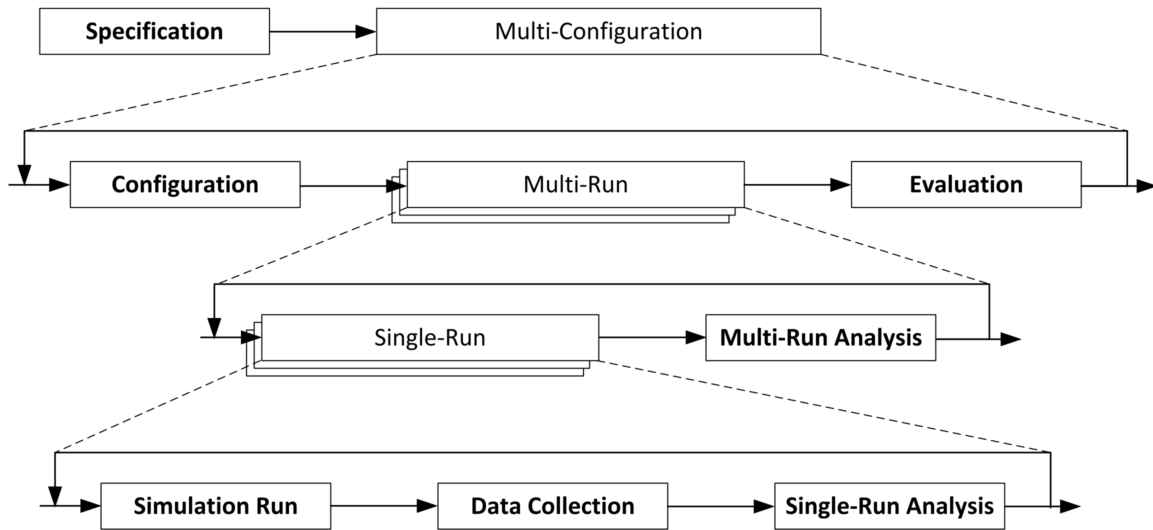



Figure 2.1: Layered view of the six tasks of a simulation experiment based on [116].

- **Simulation Run:** Perform a simulation run.
- **Data Collection:** Collect data during simulation runs. Only collect data that is needed for a proper analysis.
- **Analysis:** Analyze simulation run results in two phases. First, analyze results of a single simulation run (single-run analysis). Second, analyze the results of multiple replications.
- **Evaluation:** Produce feedback, e.g., visualizations, based on the analysis results for the configuration task to determine further interesting parameter settings.

Figure 2.1 shows the relation of these six tasks of a simulation experiment in a layered view. All tasks play an essential role to perform a successful simulation experiment. For example, even if the runtime of a simulation experiment is low, poorly chosen model configurations might reduce the possible conclusions drawn from the gained results.

To save computational time, as few simulation runs as possible should be executed to produce sufficient useful data. For example, effectively selecting model parameter settings can be done by factorial experiment designs [104, p. 656ff.] and sophisticated sampling methods like Latin hypercube sampling [124]. Moreover, the executed number of replications should be determined dynamically, e.g., by using confidence intervals [104, p. 522ff.]. Besides, sophisticated termination criteria should be used to terminate a simulation run, e.g., by using steady state analyzer [104, p. 544ff.]. Finally,

the minimum of data that is needed for the success of the simulation experiment should be collected during the simulation runs. Figure 2.2 illustrates the importance of data collection. It shows the runtime behavior of simulation runs performed with an endocytosis and endosome maturation model implemented in the modeling language ML-Rules [130] with different data collection settings that are specified with a SQL-like domain-specific instrumentation language [77]. The results emphasize that writing simulation run results to the disk can significantly increase the runtime of a simulation run. To reduce the amount of data that is written to the disk, streaming approaches can be used that directly stream observed data intelligently to components processing this data [174].

Although all tasks of a simulation experiment are crucial for its success, simulators are often in the focus of new approaches and concepts trying to improve the runtime efficiency of simulation runs. Nevertheless, other performance metrics are also of importance, e.g., the memory consumption, energy consumption, accuracy etc. In modeling and simulation, especially the accuracy is often considered as many simulators trade accuracy for runtime efficiency. The runtime performance of a simulator can be analyzed theoretically, e.g., by determining its best-case, average-case, and worst-case time complexity [109]. However, these theoretical measurements do not necessarily reflect the runtime performance of a simulator's concrete implementation. The area of experimental algorithmics deals with theoretical and empirical analysis, i.e., it is concerned with the analysis of algorithms to predict "[...] how well a given algorithm will perform in a given scenario under given conditions and assumptions".[134, p. 1]. Basically, theoretical questions are combined with empirical research methods.

The development of new simulators and the enhancement of existing simulators eventually lead to sets of simulators and configurations available for executing simulation runs. A user has to choose from the available options and therefore has to deal with the well-known *algorithm selection problem* [166], see Figure 2.3. The algorithm selection problem describes the problem to select an algorithm $a \in \mathbb{A}$ to solve a problem instance $x \in \mathbb{P}$, based on extracted problem features $f(x) \in \mathbb{F}$, user criteria $w \in \mathbb{R}^n$,a performance metric $p : \mathbb{A} \times \mathbb{P} \rightarrow \mathbb{R}^n$, and a performance weighting function changing a calculated performance based on user criteria $g : \mathbb{R}^n \rightarrow \mathbb{R}^n \times \mathbb{R}^n$. Generally speaking, a selection mapping $S : \mathbb{F} \times \mathbb{R}^n \rightarrow \mathbb{A}$ has to be determined, i.e., given some features and user criteria, determine an algorithm to solve the problem instance. Solving the algorithm selection problem is a challenging task. For example, the effectiveness of algorithm selection methods essentially depends on a suitable selection of problem features that refers to the challenging *feature selection problem* [108, 67]. Referring to modeling and simulation, a problem instance $x \in \mathbb{P}$ represents the task to execute a simulation run with a concrete model within a concrete environment. The

Query 1:

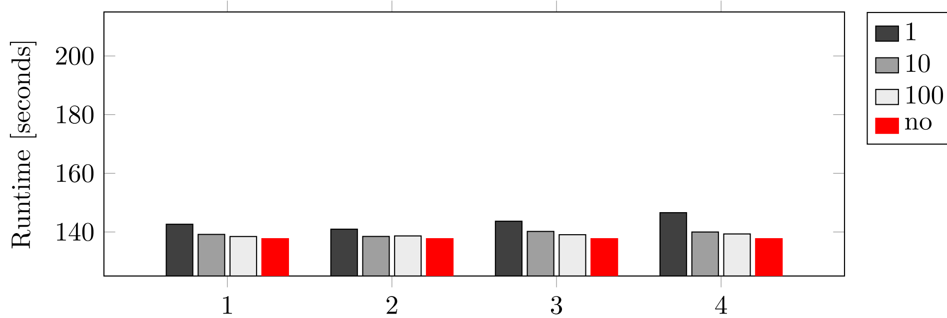
```
INSTRUMENT model OBSERVE COUNT(species.quantity) WHERE TRUE GROUP BY species.name
    EVERY n STEPS;
```

Query 2:

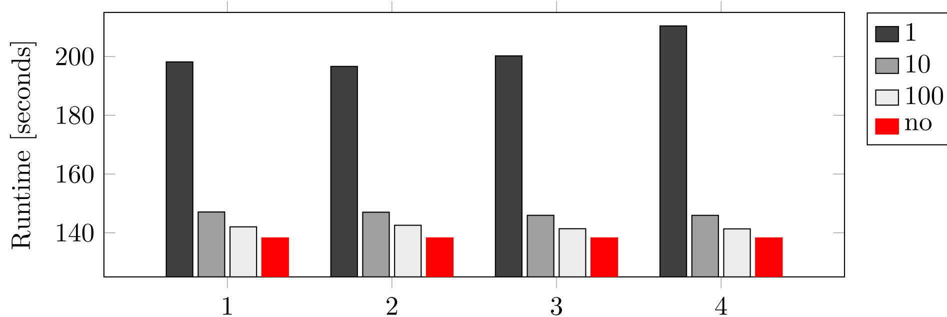
```
INSTRUMENT model OBSERVE COUNT(species.quantity) WHERE (species.name = 'Endosome' AND
  species.attribute(2) = 'late') GROUP BY species.name EVERY n STEPS;
```

Query 3:

```
INSTRUMENT model OBSERVE SUM[DOUBLE](species.attribute(1)), AVG[DOUBLE](species.
    attribute(1)),
  MIN[DOUBLE](species.attribute(1)), MAX[DOUBLE](species.attribute(1))
  WHERE species.name = 'Endosome' GROUP BY species.name EVERY n STEPS;
```




(a) Runtime of observation code without data storing.




(b) Runtime of observation code including time for direct writing of the observations to the disc.

Figure 2.2: From [77]. Test queries (top) and runtime results (bottom) of simulation runs with the endocytosis and endosome maturation model implemented in ML-Rules. Presented are the minimal runtime values for each of the setups. The 4th setup comprises all three queries in the same run. The instrumentation interval n is coded in the colors. Query 1 simply counts the amounts of all species grouped by their name (including attributes) after every $n$ simulation events. Query 2 counts the amounts of all `Endosome` species that have the second attribute value `'late'` after every $n$ simulation events. Query 3 computes the sum, minimum, maximum, and average of the first attribute value of all `Endosome` species after every $n$ simulation events. The values $\{1, 10, 100\}$ have been used for $n$. The red bars represent the runtime without any observation.

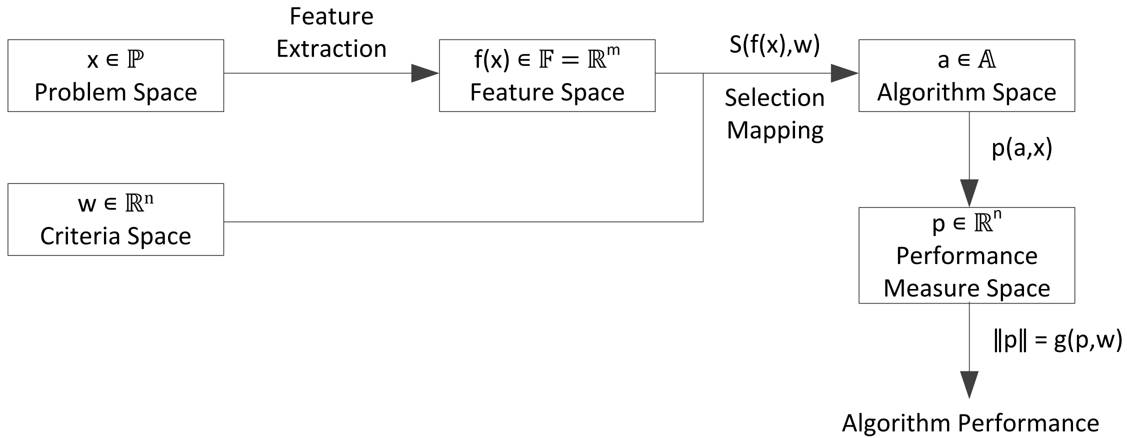



Figure 2.3: The algorithm selection problem as defined by Rice [166].

algorithm set $\mathbb{A}$ includes all available simulators with all valid configurations, i.e., each element $a \in \mathbb{A}$ represents a concrete simulator with a concrete configuration. Further, the performance metric can include various measurements like the runtime and the memory consumption to execute a simulation run or in case of approximate simulators the accuracy of the results. Finally, selected problem features can for example refer to model properties, software properties or hardware properties.

To deal with this selection challenge, concepts of adaptive software, see Definition 7, are often applied. Generally, such concepts cannot only applied in simulation software to change simulators, but also to adapt other parts of a simulation experiment, e.g., steady state estimator or replication number criteria [117]. A comprehensive adaptive simulation software would be able to adapt itself referring to all tasks of a simulation experiment. Nevertheless, in this thesis, we concentrate on the adaptation of simulators. The next four sections illustrate existing concepts and algorithms adapting simulators.

**Definition 7. Adaptive Software** Software that is able to change its behavior according to input and environmental changes is called adaptive.

## 2.2 Continuous Simulation

Models with state variables changing continuously with respect to time — the dynamics typically described in form of ordinary differential equations — can be simulated with numerical integration methods [104, p. 109]. By relying on the Taylor series, these

methods approximate solutions of initial value problems

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0$$

and calculate approximations for discrete time points $t_1$, $t_2$, $t_3$, etc. Many numerical integration methods exist, e.g., the Euler method, the classical Runge-Kutta method, or the Dormand-Prince 54 method [39]. The three mentioned methods are single-step methods that only use the approximation of $y_i$ to approximate $y_{i+1}$. As usual, no method dominates the others and the performance of a concrete method depends on the model to be simulated. Consequently, automatic selection mechanisms have been developed to relieve the user selecting methods manually. For example, Kamel et al. have developed an expert system (*ODExpert*) that analyzes a problem, e.g., its stiffness, to support the user selecting a suitable numerical integration method from a pre-defined fixed set of methods based on a decision tree [102]. Similarly, Bunus developed a framework called *ModSimPack* that also uses a decision tree to select a solver for a given problem considering several features, e.g., its stiffness and the structure of the Jacobian matrix [21]. In contrast to *ODExpert*, the framework *ModSimPack* does not only recommend a method but also applies it automatically. Another approach presented by Claeys et al. does not rely on problem features, but uses a repository of test models to evaluate available solver [32, 33]. Nevertheless, how the gained performance and accuracy knowledge shall be used to automatically select a solver is not answered in detail.

Besides such selection mechanisms, adapting the selected method itself during runtime is also an established approach for numerical integration methods. For example, the step size $h \in \mathbb{R}^+$, i.e., the interval between calculated time points, essentially influences the performance of simulation runs and the accuracy of the simulation run results. Typically, the smaller the step size, the more accurate are the results. However, a smaller step size also results in a higher computational effort since more steps have to be calculated. Further, using a fixed step size for a whole simulation run is typically unsuitable — the smallest step size that achieves sufficient accuracy for every phase of the simulation run must be selected, although it might be acceptable to apply larger step sizes for some phases. To tackle this issue, step size control mechanisms can be built on top of numerical integration methods. The basic idea of these mechanisms is to approximate the error made and increase or decrease the step size if the approximated error is high or low. Two approaches are typically distinguished: step doubling and embedded error estimation [159]. In step doubling, the approximation $y_{n+1}$, $t_{n+1} = t_n + h$ is calculated twice with the same method — once with $h$ and once with $h' = h/2$. Both approximations are compared and if the

$$
\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + \frac{1}{5}h, y_n + \frac{1}{5}k_1) \\
k_3 &= f(t_n + \frac{3}{10}h, y_n + \frac{3}{40}k_1 + \frac{9}{40}k_2) \\
k_4 &= f(t_n + \frac{4}{5}h, y_n + \frac{44}{45}k_1 - \frac{56}{15}k_2 + \frac{32}{9}k_3) \\
k_5 &= f(t_n + \frac{8}{9}h, y_n + \frac{19372}{6561}k_1 - \frac{25360}{2187}k_2 + \frac{64448}{6561}k_3 - \frac{212}{729}k_4) \\
k_6 &= f(t_n + h, y_n + \frac{9017}{3168}k_1 - \frac{355}{33}k_2 + \frac{46732}{5247}k_3 + \frac{49}{176}k_4 - \frac{5103}{18656}k_5) \\
k_7 &= f(t_n + h, y_n + \frac{35}{384}k_1 + \frac{500}{1113}k_2 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6) \\
y'_{n+1} &= y_n + \frac{5179}{57600}k_1 + \frac{7571}{16695}k_3 + \frac{393}{640}k_4 - \frac{92097}{339200}k_5 + \frac{187}{2100}k_6 + \frac{1}{40}k_7 \\
y''_{n+1} &= y_n + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6.
\end{aligned}
$$

Figure 2.4: Equations of the Domarnd Prince 54 method [39].

difference is greater (smaller) than a threshold $\epsilon$, the step size is decreased (increased). This approach can directly be applied to single-step numerical integration methods, but it does not exploit method specific information, e.g., no intermediate results needed to calculate one approximation are reused to calculate the other approximation. Reusing intermediate results is done by methods applying the embedded error estimation like the Dormand-Prince 54 method[1]. The Dormand-Prince 54 method calculates two approximations $y'_{n+1}$ and $y''_{n+1}$ as shown in Figure 2.4. Whereas the value $y'_{n+1}$ is a fifth-order approximation, i.e., the local truncation error is in the order $O(h^5)$, the value $y''_{n+1}$ is a fourth-order approximation with a local truncation error in the order $O(h^4)$. Consequently, $y'_{n+1}$ is a better approximation than $y''_{n+1}$. The absolute difference $|y'_{n+1} - y''_{n+1}|$ is interpreted as the error of $y''_{n+1}$ and used to decide whether a) the step has to be repeated with a smaller step size, b) the step size has to be decreased, or c) the step size can be increased. By reusing the terms $k_1$ to $k_6$ to calculate both approximations, the computational costs to calculate the error approximation and thus the computational costs of the step size control are comparably low.

[1] In Matlab, this method is called `ode45`. It is also available in the Apache Commons Math 3.6.1 library as `DormandPrince54Integrator`.

Although step size control can improve the performance of a simulation run with an acceptable loss of accuracy, changing the integration method itself can also be beneficial. For example, Petzold developed an algorithm that changes between a method suitable for stiff systems, e.g., systems with different time scales, and a method suitable for nonstiff systems during a simulation run based on the potential step size of both methods [158].

## 2.3 Parallel Discrete Event Simulation

The parallel discrete event simulation (PDES) partitions a model into separated logical processes (LP) that are simulated in parallel [58]. During the simulation, logical processes communicate with each other via event messages. Only event messages trigger state changes of LPs. The main challenge of PDES is to preserve the causality constraint, i.e., all executed events on an LP must be processed in time stamp order. Synchronization simulators are used to avoid causality errors. These simulators can either be conservative or optimistic. Conservative simulators like the Null Message Algorithm [29] only allow an LP to execute safe events and thus the parallelism may not be fully exploited. In contrast, optimistic simulators process unsafe events, but need rollback mechanisms to prevent incorrect results and thus may process many events unnecessarily, e.g., Time Warp [96]. Various approaches exist to realize the rollback mechanism [155], e.g., state saving methods or reverse computation methods. To improve the performance of optimistic simulators, the optimism is often restricted so that unsafe events are only allowed to be computed if additional conditions are satisfied. For example, the moving time window (MTW) protocol [181] limits the optimism of the LPs by only allowing to process events within the time interval $[GVT, GVT + \omega]$, where $\omega \in \mathbb{R}^+$ is a parameter and GVT is the global virtual time [58, p. 74].

Furthermore, adaptive methods are often applied to control the optimism of optimistic simulators, e.g., if few rollbacks are executed, the optimism can be increased and if many rollbacks are necessary, the optimism should be decreased. The MTW protocol can easily be made adaptive by controlling the time window $\omega$ dynamically during runtime [149]. The time window is adjusted after every GVT computation depending on the relation of all events and unprocessed events. The *Dynamic Local Time Window Estimates (DLTWE)* technique developed by Bauer et al. applies a similar strategy that bounds the advancement of the LPs based on time estimates of next events that are spread by the LPs [11]. Another approach is *Penalty-Based Throttling* [163], i.e., an LP is penalized every time it is executing a rollback, i.e., it gets fewer resources to compute the simulation run. Child and Wilsey follow

this idea by using *Dynamic Voltage and Frequency Scaling* of modern processors to throttle individual LPs [31]. Interestingly, their goal is not only to improve the runtime efficiency of the simulation, but also to save energy. The *Switch Time Warp* algorithm changes the prioritization of LPs to reduce the number of rollbacks [184]. The assumption is that the number of LPs is greater than the number of available processors, so that a scheduling mechanism can prefer processors that execute few rollbacks.

Ball and Hoyt introduce a blocking window between two successive event processings [8]. This blocking window is adapted after every event execution for each LP so that the time spent for blocking and doing rollbacks is minimized. Similarly, Srinivasan and Reynolds designed the *Elastic Time Algorithm (ETA)* that introduces a dynamic delay between the execution of two events [183]. The *ETA* has been extended by Quaglia to also consider real costs of rollbacks, resulting in the *Scaled Elastic Time Algorithm (SETA)* [161].

In [115], an adaptive simulator has been developed for the parallel simulation of multi-agent systems. A shared state, which represents the agents' environment, is typically used for the parallel simulation of multi-agent systems. This state is not associated with an individual LP, however, all LPs can read and write to each variable of it. Rollbacks are necessary if an LP writes a variable of the shared state at simulation time $t_i$ that has already been read by another LP at simulation time $t_i + \epsilon$. Such read operations are called premature. Basically, the approach presented by Lees et al. delays read operations depending on their probability to be premature to reduce the number of rollbacks to be executed. The same group has also presented an adaptive mechanism that dynamically distributes the shared state over the network so that variables are stored on the physical machine that executes the LPs that mostly access these variables [146]. A dynamic clustering algorithm for parallel simulation exploiting state sharing has recently been designed by Marziale et al., which clusters LPs into "groups depending on the volume of mutual state accesses along phases of the model execution" [127]. Internally, each cluster executes all events sequentially so that rollbacks are avoided within the cluster.

Another adaptive parallel simulator — in this case for network cluster simulation — is presented in [54]. Each node of the modeled cluster is simulated by an LP and messages sent represent packets sent through the network. The simulation proceeds quantum-synchronized, i.e., all LPs are regularly synchronized to receive all transient messages from the other LPs. The wallclock time length between two synchronizations is referred to as the quantum. In contrast to other PDES simulators, this simulator allows inaccuracy, i.e., if $LP_i$ sends a message with time stamp $t_i$ to $LP_j$ with the current simulation time $t_i + \epsilon$, $LP_j$ does not execute a rollback, but it simply increases

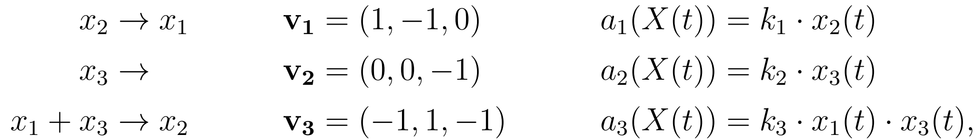
the time stamp of the received message to $t_i + \epsilon$. Generally speaking, the simulation time a packet needs from one node of a cluster to another node is simply increased by $\epsilon$. Although this approach creates an inaccuracy of the simulation results, no rollbacks are needed. Clearly, the accuracy of this approach is influenced by the length of the quantum. The algorithm developed by Falcon et al. dynamically increases or decreases the quantum depending on the number of messages that are received within one quantum.

In [182], the *Clustered Adaptive Distributed Simulator (CADS)* is presented. This simulator partitions all LPs to clusters based on various properties of the LPs, so that all LPs of one cluster are executed on the same physical machine. To reduce the number of rollbacks, cluster buffers are introduced that delay messages that shall be sent from one cluster to another cluster. For each buffer, the delay is dynamically increased or decreased depending on the amount of messages to be delayed and their relation to the global virtual time. An alternative approach is presented by Sherer et al. that dynamically clusters messages to reduce the message overhead [175]. Therefore, the focus is not to reduce the number of rollbacks. However, this strategy is particularly efficient if many messages would be sent with little content.

Load balancing algorithms, i.e., algorithms that distribute the LPs dynamically during runtime over the available computing resources can also be referred to adaptive algorithms used in PDES. The goal is to minimize the communication load and to maximize the processor load. However, the more balanced the processor load is, usually the more unbalanced is the communication load (and vice versa) [188]. This is why load balancing algorithms must find a trade-off between a balanced processor and communication load. For example, Boukerche et al. developed a load balancing algorithm for conservative algorithms that periodically analyzes the load of the available CPUs and distributes the LPs accordingly to maximize the overall load [18]. Peschlow et al. developed a dynamic load balancing algorithm for optimistic algorithms that consider the capacity of each host, the computational load and the communication load generated by each LP [156]. Two alternating cycles are introduced, one that balances the computational load and one that minimizes the communication costs. A user-defined number of seconds is waited between two adaptation cycles. The adaptive load balancing algorithm designed by Meraji et al. in [137] adds a further adaptivity level on top of load-balancing algorithms: it applies a selection mechanism that repeatedly chooses from two load-balancing algorithms: one balances the processor load and one balances the communication load. This approach uses reinforcement learning for the adaptation decision making and executes an adaptation after a user-defined number of cycles.

## 2.4 Simulation of Biochemical Reaction Networks

Simulating biochemical reaction networks is a popular approach to study such networks like gene regulation, metabolisms or pathways. The state of a model at simulation time $t$ is represented by a vector $X(t) = (x_1(t), x_2(t), \dots, x_n(t)) \in \mathbb{N}^n$ that describes the amount values of considered entities, e.g., the numbers of specific proteins. The dynamics of the model are defined by a set of reactions $R = \{R_1, R_2, \dots, R_m\}$. A reaction $R_i$ is described by a change vector $\mathbf{v_i} = (v_1, v_2, \dots, v_n) \in \mathbb{Z}^n$ and a propensity function $a_i : \mathbb{N}^n \rightarrow \mathbb{R}^+$. A biochemical reaction network can be simulated deterministically by converting the reaction network to a set of ordinary differential equations (ODEs) [113]. For example, given the initial state vector $X(t_0) = (x_1(t_0), x_2(t_0), x_3(t_0))$ and the reaction network



$$
\begin{aligned}
x_2 &\rightarrow x_1 & \mathbf{v_1} &= (1, -1, 0) & a_1(X(t)) &= k_1 \cdot x_2(t) \\
x_3 &\rightarrow & \mathbf{v_2} &= (0, 0, -1) & a_2(X(t)) &= k_2 \cdot x_3(t) \\
x_1 + x_3 &\rightarrow x_2 & \mathbf{v_3} &= (-1, 1, -1) & a_3(X(t)) &= k_3 \cdot x_1(t) \cdot x_3(t),
\end{aligned}
$$

the resulting ODEs would look like follows

$$
\begin{aligned}
\frac{dx_1}{dt} &= k_1 \cdot x_2(t) - k_3 \cdot x_1(t) \cdot x_3(t) \\
\frac{dx_2}{dt} &= k_3 \cdot x_1(t) \cdot x_3(t) - k_1 \cdot x_2(t) \\
\frac{dx_3}{dt} &= -k_2 \cdot x_3(t) - k_3 \cdot x_1(t) \cdot x_3(t).
\end{aligned}
$$

This set of ODEs can be simulated deterministically with numerical integration methods, see Section 2.2. In this case, species populations are continuous, i.e., $X(t) \in (\mathbb{R}^+)^n$. However, in this case stochastic effects are not considered that become important in case of small entity numbers. Based on the chemical master equation, biochemical reaction networks can be simulated stochastically by interpreting them as continuous-time Markov chains (CTMC) and applying the stochastic simulation algorithm (SSA) [63]. The basic algorithm (*Direct Method*) is described in Algorithm 2.1.

Many variants of this algorithm exist also producing exact results. For example, the *First Reaction Method* computes the firing time for each reaction individually by sampling numbers from exponential distributions with their propensities as rates and executing the reaction with the lowest firing time next [63]. Afterward, the propensities and firing times of all reactions are updated accordingly. To improve this algorithm, Gibson and Bruck developed the *Next Reaction Method*, which firstly uses a priority queue to store the firing times and by using a *dependency graph*

**Algorithm 2.1** Sketch of a simulation step in the basic SSA (Direct Method [63]).
$X(t_j) = (x_1(t_j), x_2(t_j), \ldots, x_n(t_j)) \in \mathbb{N}^n$: the state vector at time $t_j$.
$R = \{R_1, R_2, \ldots, R_m\}$: the set of reactions.
$\mathbf{v_i} \in \mathbb{Z}^n$: state change vector of reaction $R_i$.
$a_1(X(t_j)) \ldots a_m(X(t_j))$: propensities of the reactions $R_1, \ldots R_m$.
$a_0(X(t_j))$: propensity sum of all reaction propensities.

```
1  // Calculate propensity sum of all reactions
2  a_0(X(t_j)) := ∑_{i=1}^m a_i(X(t_j))
3
4  // Select a reaction to be executed, see Algorithm 2.2
5  i := select(R, a_0(X(t_j))))
6
7  // Advance simulation time by sampling a number from an exponential
8  // distribution with rate λ = a_0(X(t_j))
9  t_{j+1} := t_j + Exp(a_0(X(t_j)))
10
11 // Execute selected reaction
12 X(t_{j+1}) = X(t_j) + v_i
```

**Algorithm 2.2** Algorithmic selection of a reaction in the SSA. The probability $P(R_i)$ to select reaction $R_i$ is its relation of its propensity to the propensity sum: $P(R_i) = \frac{a_i(X(t_j))}{a_0(X(t_j))}$.
$a_1(X(t_j)) \ldots a_m(X(t_j))$: propensities of the reactions $R_1, \ldots R_m$.
$a_0(X(t_j))$: propensity sum of all reaction propensities.

```
1  sum := 0
2  x := U(0, a_0(X(t_j)))
3
4  for (i ∈ (1, ..., m)) {
5    sum := sum + a_i(X(t_j))
6    if (sum > x) {
7      return i
8    }
9  }
```

secondly updates only those reaction propensities after a reaction execution that are influenced by the currently executed reaction [62]. Similarly, the *Optimized Direct Method* updates propensities only if necessary, but additionally also sorts the calculated propensities based on precalculated reaction frequencies. Sorting propensities can improve the efficiency of the reaction selection method [27], see Figure 2.5. However, the propensities are not sorted based on their actual values directly, since that would cause too much computational effort during runtime. The *Sorted Direct Method* enhances the sorting idea by introducing an adaptive sorting scheme that calculates

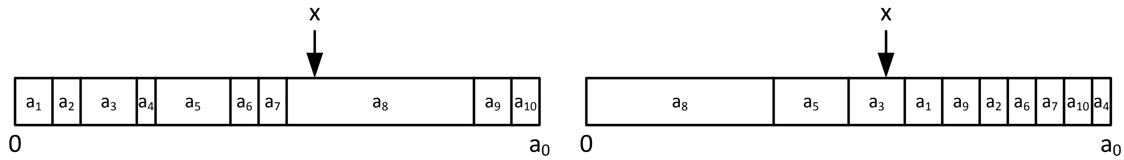



Figure 2.5: Selecting a reaction based on Algorithm 2.2 is more efficient when the reactions' propensities are sorted [27]. In the given example, eight iterations would be needed for the unsorted case (assuming that propensities are considered from left to right) to find the first propensity so that the sum of all considered propensities is greater than $x$, whereas only three iterations would be needed for the (perfect) sorted case. Although the concrete selected reaction differs, the probability to select a reaction does not change.

the reaction frequencies during runtime and does not need any precalculations [133]. A last approach that shall be presented is the recently developed *Rejection-based SSA* that reduces the number of propensity updates by computing propensity upper bounds and lower bounds and calculating concrete propensity values only when needed [189].

So far, all presented simulators are variants of the basic SSA computing exact results. None of these methods dominates the others, i.e., the performance of the methods and also the best performing method depend on the model to be simulated [128]. Thus, this application field is suitable for applying algorithm selection methods. In [47], Ewald applied a generic simulator selection mechanism implemented with the simulation algorithm selection framework (SASF), see Section 2.5, for selecting exact SSA simulators automatically.

### 2.4.1 $\tau$-leaping

Besides these exact simulators, approaches exist that trade accuracy for runtime efficiency analog to numerical integration methods and therefore these approaches compute only approximations of the exact results. A common approximate SSA variant is $\tau$-leaping, which performs "leaps" along the time line executing many reactions simultaneously during each leap [64, 25, 78]. Basically, one leap can be described by

$$X(t+\tau) = X(t) + \sum_{i=1}^{m} \mathbf{v_i} \cdot Pois(a_i(X(t)) \cdot \tau), \tag{2.1}$$

where $\tau$ is the interval of the leap and $Pois(a_i(X(t)) \cdot \tau)$ is a number of firings for reaction $R_i$ sampled from a Poisson distribution with rate $a_i(X(t)) \cdot \tau$.

**Definition 8. Leaping Condition** Require $\tau$ to be small enough that the change in the state during $[t, t+\tau]$ will be so slight that no propensity function will suffer an appreciable (depending on an error parameter $\epsilon \in [0,1]$) change in its value [64].

Analog to step size control mechanisms used for numerical integration methods, $\tau$-leaping is adapting $\tau$ during a simulation. In principle, it would be possible to apply *step doubling* (see Section 2.2) also for $\tau$-leaping. However, the *leaping condition* is applied instead, see Definition 8. For each leap, the largest $\tau$ that does not violate the leaping condition shall be applied. A well-known variant to calculate $\tau$ has been developed by Cao et al. [25] and works as follows (current simulation time is $t_j$). Before each leap, the reaction set $R$ is divided into non critical reactions $R_{ncr}$ and critical reactions $R_{cr}$, i.e., $R = R_{ncr} \cup R_{cr}$. A reaction is a non critical reaction if it can be fired more often than $n_c \in \mathbb{N}$ times; otherwise it is a critical reaction. Whereas multiple non critical reactions are allowed to be executed several times during one $\tau$-leap, at most one critical reaction is executed once during one $\tau$-leap. This is done to reduce the probability to get negative species amounts due to the unbound Poisson distribution. Next, a candidate $\tau'$ is computed based on the non critical reactions as follows:

$$\tau' = \min_{s \in RS_{ncr}} \left\{ \frac{\max\{\epsilon \cdot X(t_j)_s / g_s, 1\}}{|\hat{\mu}_s|}, \frac{\max\{\epsilon \cdot X(t_j)_s / g_s, 1\}^2}{\hat{\sigma}_s^2} \right\}. \tag{2.2}$$

The set $RS_{ncr}$ contains all reactants of all non-critical reactions. The value $g_s$ is used to "guarantee that bounding the relative change of states is sufficient for bounding the relative change of propensity functions" [173] and is computed as follows:

$$g_s = h_s + \frac{h_s}{n_r} \sum_{i=1}^{n_s - 1} \frac{i}{X(t_j)_s - i}, \tag{2.3}$$

whereby $h_s$ denotes the highest order of all reactions in $R_{ncr}$ the species $s$ is involved and $n_s$ denotes the highest amount of the species $s$ which is consumed in any of these reactions. The variables $\hat{\mu}_s$ and $\hat{\sigma}_s^2$ represent the mean and the variance of the change of $s$ for all reactions in $R_{ncr}$ in the current state:

$$\hat{\mu}_s = \sum_{r \in R_{ncr}(X(t_j))} \mathbf{v_{r,s}} \cdot a_r(X(t_j)) \qquad \hat{\sigma}_s^2 = \sum_{r \in R_{ncr}(X(t_j))} \mathbf{v_{r,s}}^2 \cdot a_r(X(t_j)) \tag{2.4}$$

If the computed $\tau'$ is smaller than $\alpha \cdot a_0(X(t_j)), \alpha \in \mathbb{N}$, the firing numbers of most reactions would probably be 0. Therefore, no leap is executed in this case but $N_{SSA} \in \mathbb{N}$ steps of the normal SSA are executed instead. If $\tau'$ is sufficiently large, a second candidate $\tau''$ using the critical reactions is calculated that represents the next

firing time of a critical reaction:

$$\tau'' = Exp(\sum_{cr \in R_{cr}} a_{cr}(X(t_j))) \tag{2.5}$$

The minimum of $\tau'$ and $\tau''$ is used to calculate the leap as described in Equation 2.1. Additionally, if $\tau'' < \tau'$, one critical reaction is selected analogously to the reaction selection of the SSA, which will also be executed. If any negative population occurs after executing the reactions, all changes are discarded, $\tau'$ is halved and the procedure is repeated until a valid $\tau$-leap is executed. By introducing critical reactions executed in an SSA manner — a feature not integrated into the initial version of $\tau$-leaping in [64] — the simulator basically performs an adaptive model separation into two parts. The *maximal time step method* presented by Puchałka and Kierzek follows the same idea as the $\tau$-leaping variant of Cao presented in [25], but makes the separation between $\tau$-leaping and the SSA explicit as they refer to their algorithm as an "approach combining the Gibson and Bruck algorithm with the Gillespie $\tau$-leap method" [160]. Altogether, $\tau$-leaping as described by Cao et al. in [25] is adaptive in several ways. First, it adapts the step size each step. Second, it combines the original $\tau$-leaping with an SSA and decides to execute the SSA if the calculated $\tau$ is too small. Third, the model is dynamically separated into critical reactions (simulated exactly) and noncritical reactions (simulated approximately).

The presented $\tau$-leaping method is explicit, i.e., only $X(t_j)$ is considered to approximate $X(t_j + \tau)$. Analog to the deficiencies of explicit numerical integration methods for stiff systems, explicit $\tau$-leaping is not suited to simulate stiff biochemical reaction networks as it would perform tiny leaps. Thus, Rathinam et al. have developed an implicit variant of $\tau$-leaping in [162]. Computing a leap with the implicit $\tau$-leaping is more expensive, so that it should only be used for stiff systems, whereas the explicit $\tau$-leaping shall be used for nonstiff systems. Consequently, analog to the switching approach developed by Petzold in [158] to switch between numerical integration methods suitable for stiff or nonstiff systems, an adaptive $\tau$-leaping algorithm has been developed that dynamically switches between the explicit and the implicit $\tau$-leaping during runtime [26, 173].

### 2.4.2 Combining Discrete and Continuous Simulators

Referring to the simulation of biochemical reaction networks, a common strategy is to combine stochastic and continuous approaches, often referred to as *hybrid* simulators. For example, Haseltine and Rawlings present a simulator that initially partitions the set of reactions into *fast* and *slow* reactions ($R^f$ and $R^s$) depending on their

**Algorithm 2.3** Sketch of a simulation step in the hybrid simulator presented in [71].
$t_j$: current simulation time after $j$ simulation steps,
$R^s = \{R_1^s, \dots R_{m_s}^s\}$: slow reaction set, $R^f = \{R_1^f, \dots R_{m_f}^f\}$: fast reaction set
$a_1^s \dots a_{m_s}^s$: propensities of slow reactions,
$\tilde{X}(t_j)$: intermediate state of the system after integrating the fast reactions and before executing a slow reaction.

```
1  // Calculate propensity sum of slow reactions
2  a_0^s(X(t_j)) := \sum_{i=1}^{m_s} a_i^s(X(t_j))
3
4  // Select a slow reaction to be executed based on the slow
5  // reaction set R^s, see Algorithm 2.2
6  i := select(R^s, a_0^s(X(t_j)))
7
8  // Calculate a simulation time advance by sampling a number
9  // from an exponential distribution with rate \lambda = a_0^s(X(t_j))
10 \tau := Exp(a_0^s(X(t_j)))
11
12 // Integrate fast reactions until t_j + \tau
13 \tilde{X}(t_j) := integrate(X(t_j), R^f, \tau)
14
15 // Advance simulation time
16 t_{j+1} := t_j + \tau
17
18 // Execute selected slow reaction
19 X(t_{j+1}) := \tilde{X}(t_j) + \mathbf{v_i^s}
```

propensities [71]. When models contain reactions with propensities differing by several orders of magnitude, a purely stochastic simulator would spend most of the runtime to execute firings of the fast reactions. The idea of the simulator by Haseltine and Rawlings is to approximate these fast reactions deterministically by using numerical integration methods and to only calculate the slow reactions stochastically by using the exact SSA. A simulation step is sketched in Algorithm 2.3. The partitioning of fast and slow reactions is done once at the beginning of a simulation run by using heuristics considering the reactions and initial species amounts. Since slow reaction propensities are probably dependent on species that are changed by fast reactions, this approach is approximate. In [71], Haseltine and Rawlings themselves propose a *probability of no slow reaction* to decrease the step size $\tau$ if necessary.

In general, many further hybrid simulators exist, which work similarly like the presented one. For example, besides reaction propensities, species amounts are also often considered while partitioning the reactions, i.e., a fast reaction should not change a species with a small amount [171]. Another approach is to separate reactions in terms of species components, as using propensities and species amounts do “not always

provide a convenient description of the hybrid stochastic process" [35]. Finally, many hybrid simulators adapt the partitioning during runtime to respond to significant propensity changes. For example, the simulator presented by Herajy and Heiner performs a repartitioning if the propensity sum of all reactions leaves a user-defined interval or the amount of a species drops below a user-defined threshold [83]. The simulator presented in [112] dynamically splits species into discrete and continuous species depending on their amount, i.e., if the amount drops below a threshold $t_{low}$, a species is classified as discrete and if the amount exceeds a second threshold $t_{high}$ ($t_{high} - t_{low} > K$, e.g., $K = 100$), a species is classified as continuous. The two thresholds $t_{low}$ and $t_{high}$ are used to avoid rapid changes of the species classification. A reaction is a fast reaction only if all reactants refer to continuous species. In [148], Pahle analyzes several hybrid simulators referring to their partitioning scheme and applied continuous and stochastic simulators.

The hybrid simulators presented in [24] and [40] do not integrate the fast reactions directly. Instead, they assume that stationary distributions exist for all species involved in fast reactions that are either computed analytically ([25]) or empirically ([40]). These distributions are used to update the propensities of slow reactions, which are simulated as usual using the basic SSA. Although this approach avoids integrating the fast reactions, it can only be applied if 1) fast reactions result in stationary distributions for all involved species of the fast reactions and 2) these distributions are reached fast, i.e., much faster than the next firing time of a slow reaction.

Hybrid simulators are also applied for spatial heterogeneous biochemical systems. In [55], Ferm et al. present an adaptive variant of the *Next Subvolume Method* (NSM)[42]. Basically, the NSM partitions the system space into artificial subvolumes ensuring spatial homogeneity for each subvolume and adds diffusion reactions to the system to enable entities to move between the subvolumes. The simulator developed by Ferm et al. extends the NSM by using either a deterministic, an approximate or an exact method to calculate the diffusion processes depending on the amounts of diffusing species. Another hybrid spatial simulator is the *Two-Regime Method*, which partitions the space into compartment-based and molecular-based areas [56]. Due to the partitioning, suitable diffusion reactions are added for entities to change between the two regimes. The partitioning of the space must be defined by the user and is fixed throughout the simulation.

## 2.5 Simulator Selection for Discrete Event Simulation

The adaptive means used by the simulators presented in the previous sections are tailored to specific application scenarios or to simulators that are extended by these means. For example, the *step doubling* method can only be applied to approximate simulators like single step numerical integration methods. Further, selection mechanisms like *ODExpert* can only be applied to numerical integration methods. The adaptive means of most of the parallel simulators presented in Section 2.3 are direct extensions of existing simulators and are tailored to parallel optimistic discrete event simulation. Some concepts are even only applicable in case of specific models to be executed like the adaptive simulator developed in [54], which can only be used to simulate network clusters in parallel. The $\tau$-leaping algorithm is another example for a simulator with a specific adaptation scheme particularly developed for it. Clearly, the advantage of these tailored mechanisms is that properties of the application scenario or the extended simulator can be exploited and the adaptive scheme can be tailored by the developer to improve its effectiveness.

A more general approach has been explored by Ewald et al. [49]. The simulation algorithm selection framework SASF has been developed, which can be used to automatically select simulators for the execution of simulation runs. It is integrated into the plugin-based modeling and simulation framework JAMES II [87] and it is developed as a framework to "prescribe how the different [selection] techniques interact with the host system, thereby hiding the internal complexity of the overall task as much as possible" [47, p. 140]. The SASF is neither restricted to a specific simulator nor to any modeling paradigm. The following section 2.5.1 describes how simulators can be represented generically in JAMES II. In the following sections 2.5.2 and 2.5.3, both simulator selection approaches within the SASF are presented.

### 2.5.1 Simulators as Selection Trees

JAMES II is a plugin-based modeling and simulation framework written in Java, which provides means to support the development of modeling formalisms and simulators and means to support the experimentation with these [87]. It uses a plugin-based scheme to follow the separation of concerns paradigm and to support a flexible architecture. Figure 2.6 shows the core packages of JAMES II— the distance of a user from the packages is thereby depicted by the layers. A strict separation between concrete models and simulators is inherently supported by JAMES II. A registry — implemented as a singleton [60, p. 144] — is responsible for the management of the plugin system

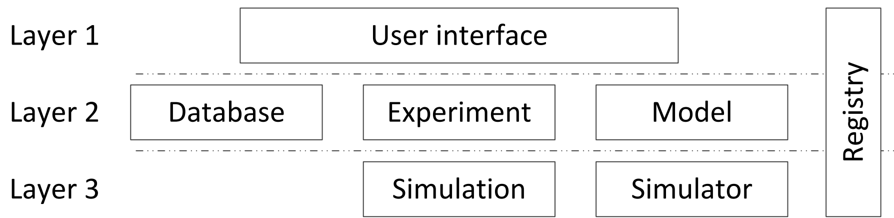



Figure 2.6: The main packages of JAMES II as described by Himmelspach and Uhrmacher [87].

of JAMES II. This concept of a central registry is common in component-based software architectures, e.g., OSGi [2]. For every component of JAMES II that shall be represented by a plugin, e.g., algorithms or data structures, a plugin type has to be created. Every implemented plugin is assigned to exactly one plugin type. Following the *factory method pattern* [60], plugins are created by factories that encapsulate the concrete instantiation of plugins. Moreover, following also the *abstract factory pattern* [60] for the selection of a concrete plugin, an abstract factory is created for each plugin-type that selects a concrete factory that creates a concrete plugin.

Parameters for plugins are defined by *parameter blocks*, which are hierarchically nested parameter structures. Every node of a parameter block contains exactly one value and a map of sub parameter blocks grouped by identifier. Especially for the configuration of hierarchical plugins in JAMES II, parameter blocks offer the needed flexibility to configure them as detailed as necessary, see Figure 2.7. When calling the plugin creation method of a factory, it also receives a parameter block containing firstly information for all primitive parameters needed to create the plugin, e.g., the size of a container, and secondly information for all sub plugins that are used by the plugin to be created. To create the sub plugin, the value of the corresponding sub parameter block contains the full qualified class name of the factory to use — by using this name, a factory object of the according class is retrieved from the registry and used to create the sub plugin given the sub parameter block to configure this sub plugin.

Generally, parameter blocks can be used to describe simulator configurations, which can be complex combinations and configurations of various plugins. Ewald introduces the notion of *selection trees* for parameter blocks describing such simulator configurations [47, p. 159]. For a simulator selection mechanism in JAMES II, the algorithm set $\mathbb{A}$ can be represented as a set of selection trees that are applicable to solve the given problem, i.e., to execute a simulation run with a given model. By using the registry of JAMES II and filtering criteria for each plugin type to be used, all applicable simulator structures to execute a simulation run can be generated

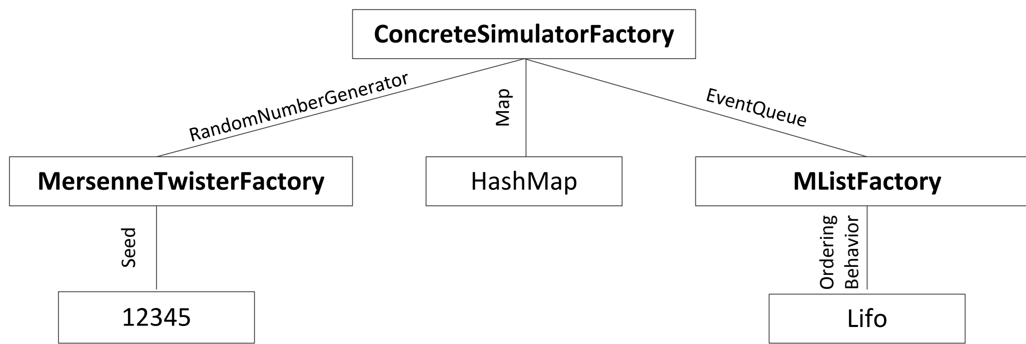



Figure 2.7: An example of a parameter block. Each node contains a value and an optional set of sub parameter blocks grouped by string identifier. Factories are used to create and configure sub plugins.

during runtime automatically. Further, by using the default values for primitive parameters, one valid parameter mapping for each vertex of the selection trees can be generated. The usage of an automatic simulator selection mechanism is motivated by large algorithm sets $\mathbb{A}$ resulted from the possible combinations of the available plugins. For example, for the component-based implementation of the simulator for the modeling language ML-Rules [130] in JAMES II (Version 0.9.7), more than 1400 valid selection trees can be generated.

### 2.5.2 Simulator Selection via Supervised Learning

The SASF comprises a performance database. Assuming that this database is filled with comprehensive performance data from problems that have been fully explored, supervised machine learning techniques can be used to generate simulator selection mappings, i.e., the ASP shall be solved by these techniques. The SASF is not restricted to one specific method, but various techniques have already been integrated, e.g., WEKA [70] to learn decision trees or JOONE [126] to apply neural networks. For each integrated technique, a selector generator is implemented within the SASF. Concrete simulator selectors are generated by these selector generators using available performance data. Since a set of selector generators can be used, naturally the question arises which one generates the best simulator selectors. This problem can in turn be interpreted as an algorithm selection problem and solved via learning methods. *Meta-Learning* methods can avoid this recursive application of machine learning methods [194]. To evaluate the performance of selectors, several mechanisms have been realized, e.g., cross validation [201, p. 152ff.] or a boolean measure of misprediction [118].

In general, the quality of generated selectors essentially depends on the available

performance data and problem features. To produce sufficient performance data for the selector generators, Ewald proposes to apply automated runtime performance exploration techniques, e.g., using synthetic benchmark models to automatically explore the problem space and gain suitable performance data, see [47, p. 225ff.]. The produced performance data can then be evaluated automatically [52]. Problem features are essential to identify relations between concrete problems and to conclude about their performance behavior, see the *feature selection problem* (Section 2.1, page 10). In the SASF, no feature selection techniques have been integrated so far.

### 2.5.3 Simulator Selection via Unsupervised Learning

As discussed in Section 2.5.2, supervised learning methods for simulator selection depend on past performance data that can be analyzed to create suitable algorithm selection mappings. In contrast, unsupervised learning methods do not rely on such performance data. Ewald et al. show in [50] how multi-armed bandit policies (MABP) [7] can be used for the simulator selection.

The regret of the MABPs is of particular interest, i.e., the relative overhead induced by exploration and sub-optimal decisions compared to an optimal policy. The regret of *zero-regret policies* converges to zero over time [193]. For example, the $\epsilon$-greedy policy is not a zero-regret policy as it chooses for any state $s$ the — based on its current knowledge — best action $a$ with the constant probability $p(s, a) = 1 - \epsilon$ and otherwise a random action. In contrast, the $\epsilon$-decreasing policy is a zero-regret strategy. This policy couples the probability $p(s, a)$ to select the best action $a$ for a state $s$ on the number $n(s)$ of occurrences of $s$: $p(s, a) = 1 - \min(1, \frac{\epsilon}{n(s)})$. Besides, the learning speed of a policy, which determines how fast the decisions improve with the number of decisions done, is also of great significance in practice [6]. However, many policies choose each action once before applying heuristics for the selection. Such policies cannot be used if the agent can choose from an infinite number of actions, e.g., if actions comprise continuous parameters.

The selection method developed by Ewald et al. using the MAPBs is called `AdaptiveSimulationRunner`. It is an extension of the `ParallelSimulationRunner` of JAMES II that manages a parallel execution of replications. The selection of a simulator for the execution of a replication is interpreted as a decision of the MABP and the performance of the simulator is used to improve the MABP's decisions. The feature selection problem is avoided as the selection method does not consider model or environment properties for the algorithm selection. Thus, for each batch of replications, the policy has to start from scratch and cannot reuse any gained knowledge. To apply an MABP, two further issues have to be considered: First, replications are typically executed in parallel, i.e., the MABP must be able to make decisions without

the feedback of all previously made decisions. Second, the MABP must distinguish between bad and faulty options, i.e., faulty options should be quarantined. Fortunately, both requirements can be easily integrated into common MABPs.

To measure the effectiveness of an MABP $p$, Ewald et al. suggest to use a metric called *relative overhead* $o_n^p$ for benchmark problems for which the performance of the best available option is known [50]. The relative overhead measures the relative performance overhead by using $p$ for $n$ replications instead of the best option constantly. It is calculated by

$$o_n^p = \frac{\sum_{i=1}^{n} reward_p^i}{n \cdot reward_{opt}} \tag{2.6}$$

where $reward_p^i$ represents the reward, e.g., executed events per second, that policy $p$ has achieved by its $i$-th decision and $reward_{opt}$ represents the expected reward of the best option. Using the relative overhead to measure the performance of a policy considers that it is not only important to find the best option in the long run, but — as the number of replications to be executed is limited — also that the learning speed of the policy is of significance.

Besides the learning speed of the MABP itself, the efficiency of this selection mechanism essentially depends on the number of available algorithms ($|\mathbb{A}|$) and the number of replications to be executed. The more replications have to be executed, the more can be explored to determine the best-performing option. Further, the fewer options are available, the faster the best-performing option will be determined. Whereas the number of needed replications cannot be influenced, the number of options can be decreased by using algorithm portfolios. In [51], Ewald et al. present an approach for the selection of simulator portfolios by genetic algorithms. The algorithm works as follows. An individual of a generation represents a specific algorithm portfolio. Each element of $\mathbb{A}$ is assigned to a unique number. The genome of individuals is represented by a list of numbers representing the elements of $\mathbb{A}$. Thus, the length of the genome determines the maximum size of the portfolio which has to be set manually. Empty slots can be used to consider portfolios with smaller sizes. The fitness of an individual is calculated by using performance data of a set of known problem instances. Assuming that the MABP will eventually find the best-performing option for a problem instance, this best-performing option is used for a problem instance and an individual to calculate its performance for this problem instance. Since the portfolio calculation shall be performed before the simulation runs of a simulation experiment, the runtime to compute a suitable portfolio has to be considered and therefore the number of calculated generations is limited. Finally, the best performing individual from the last generation is used for simulation runs of the concrete simulation experiment. The

portfolio is not changed during the execution of a simulation experiment. This can be disadvantageous if the past performance data used to generate the portfolio do not suitably reflect the actual performance features of the replications to be executed. In [73], we developed a prototypical dynamic mechanism that creates and adapts an algorithm portfolio during the execution of a simulation experiment by observing the runtime performance. Although we achieved promising results with artificial benchmark models and small numbers of algorithms ($\leq 100$), the approach has not been evaluated in more complex and realistic scenarios yet.

## 2.6 Summary

Modeling and simulation offer many opportunities to apply adaptive algorithms. Although a simulation experiment comprises various tasks that are essential for its success, adaptive approaches are typically applied to improve simulators for a better runtime efficiency of simulation runs. The techniques presented in the previous sections differ in various properties. Table 2.1 illustrates these differences for the following properties:

1. Considered Features / Measurements: What features and measurements are used for the adaptation decisions (model properties, simulator properties, environment properties, performance).
2. Adapted Property: primitive parameters (e.g., thresholds), complex parameters (e.g., partitioning of LPs), simulator.
3. Trigger & Frequency: initialization (i.e., one adaptation during the initialization of a simulation run), interval, conditional.
4. Quality Change: Whether adaptations change the quality of simulation results.

Referring to considered properties for the adaptation process, all kinds of properties are used, i.e., model, simulator, and environment properties and performance. For example, $\tau$-leaping uses the model states to calculate suitable step sizes. In contrast, *Penalty-Based Throttling* does not consider model states or model properties explicitly, but only the execution of rollbacks influences the adaptation decisions. The conservative dynamic load balancing algorithm developed by Boukerche in [18] considers the CPU loads, i.e., environmental properties, to make adaptation decisions. However, although the methods are applied to improve the runtime performance of simulation runs, in fact only few of them observe the actual performance of a simulator to evaluate whether the adaptations are beneficial or not. For example, the *Supervised Simulator Selection*

Table 2.1: Comparison of the presented adaptive means used by simulators.

| Simulator / Framework / Technique | Considered Features / Measurements | Adapted Property | Trigger & Frequency | Quality Change? |
|---|---|---|---|---|
| *Step Doubling* and *Embedded Error Estimation* [159] | model properties | primitive parameter | interval | yes |
| *ODExpert* [102], *ModSimPack* [21], Automatic Solver Selection [33, 32] | model properties, performance | simulator | initialization | yes |
| LSODA [158] | model and simulator properties | simulator | interval | yes |
| *Adaptive Moving Time Window* [149], *Local Time Window Estimates* [11] | model and simulator properties | primitive parameter | conditional | no |
| *Penalty-Based Throttling* [163] | simulator properties | complex parameter | conditional | no |
| *Switchting Time Warp* [184] | simulator properties | complex parameter | conditional | no |
| *Core Frequency Adjustment* [31] | simulator properties | primitive parameter | interval | no |
| *Adaptive Blocking Window* [8], *ETA* [183], *Scaled ETA* [161] | simulator properties, performance | primitive parameter | interval | no |
| Adaptive Throttling [115] | simulator properties | primitive parameter | interval | no |
| Adaptive Load Management [146] | model properties | complex parameter | conditional | no |
| *Granular Time Warp Objects* [127] | model properties | complex parameter | conditional | no |
| *Adaptive Network Cluster Simulator* [54] | model properties | primitive parameter | interval | yes |
| *Clustered Adaptive Distributed Simulator* [182] | simulator properties | primitive parameter | conditional | no |
| *Adaptive Message Clustering* [175] | model properties | primitive parameter | conditional | no |
| Conservative Dynamic Load Balancing [18] | environment properties | complex parameter | interval | no |
| Optimistic Dynamic Load Balancing [156, 137] | environment properties | complex parameter | interval | no |
| Sorted Reactions [27] | model properties | complex parameter | initialization | no |
| Adaptive Sorted Reactions [133] | model properties | complex parameter | interval | no |
| $\tau$-leaping [64, 25, 78] | model properties | primitive parameter | interval | yes |
| Extended $\tau$-leaping [25], *Maximal Time Step Method* [160] | model properties | primitive parameter, complex parameter and simulator | interval | yes |
| Implicit-Explicit Adaptive $\tau$-leaping [26, 173] | model and simulator properties | simulator | interval | yes |
| Hybrid Simulator for Biochemical Networks (1) [71] | model properties | complex parameter | initialization | yes |
| Hybrid Simulator for Biochemical Networks (2) [83, 171, 112] | model properties | complex parameter | conditional | yes |
| Dynamic Reaction Partitioning, e.g., [24, 40] | model properties | complex parameter | interval | yes |
| Adaptive Reaction-Diffusion Execution [55] | model properties | complex parameter | interval | yes |
| Two-Regime Method [56] | model properties | simulator | conditional | yes |
| Supervised Simulator Selection [49] | model properties, simulator properties, environment, performance | primitive parameter, complex parameter and simulator | initialization | no |
| Unsupervised Simulator Selection [49] | performance | primitive parameter, complex parameter and simulator | initialization | no |

and the *Unsupervised Simulator Selection* consider the runtime performance [47]. The considered properties are always simple to be calculated — an expected characteristic since complex calculations would reduce the effectiveness of the adaptation process.

The adaptation process itself is often also simple to be executed, i.e., most adaptive simulators change primitive parameters like thresholds, delays, or the step size. Simulator changes are rarely used by the adaptation schemes. Referring to PDES, adaptations are either be executed locally by each LP or globally by a global adaptation instance. Similarly, some simulators for biochemical reaction networks partition the set of reactions and apply one simulator for each partition and adapt this simulator. The spatial adaptive NSM simulator presented in [55] uses local adaptation decisions for each grid of the model space.

The adaptation frequency is mostly fixed and user-defined, e.g., often an adaptation

is triggered after each simulation step execution. None of the presented approaches perform an adaptation *during* the execution of a simulation step. Some approaches apply only one adaptation that is executed during the initialization of a simulation run. Besides, in some cases, it might even be easier to apply adaptations during runtime than initially, e.g., if problem features are difficult to be calculated before executing a simulation run but simple to be observed during a simulation run. For example, load balancing algorithms can observe the load of used processors during runtime to adjust the LP partitioning. Calculating according data about the potential load during a simulation run before executing it might be more difficult.

Some simulators change the accuracy of the simulation results due to adaptations, e.g., the adaptive concepts for numerical integration methods and the approximate simulators for biochemical reaction networks. If the quality of the results are changed by adaptations, developers must provide a method to approximate or restrict the error and the adaptation process must consider the potential error made by adaptations so that simulation results have always an acceptable accuracy. If no restriction is given, there is a risk that the simulator will trade too much accuracy to achieve a better runtime performance, but makes eventually the whole simulation run execution useless.

Generally speaking, besides applying individual adaptive means, in principle these techniques can often also be combined. For example, a generic method that selects a simulator during the initialization of a simulation run could automatically analyze a PDES model and decides which optimistic simulator to use. The selected optimistic simulator itself can be adaptive and could locally adapt parameters influencing the optimism of the LPs. Orthogonally to the adaptations executed by the simulator, a generic adaptation scheme could be applied on top of the simulator to adapt properties that are not adapted by the simulator itself and that are not changing the execution semantics, e.g., auxiliary data structures and sub algorithms.

All in all, most presented adaptive means are tailored to a specific application scenario and apply adaptations during runtime. However, also more generic selection methods like the methods integrated into the SASF exist and showed to be effective. Nevertheless, these methods apply an adaptation only once during the initialization of a simulation run. Altogether, a method is missing that is generic like the SASF and that supports runtime adaptations. On the one hand, the great variety of existing methods applying adaptivity during the execution of a simulation run emphasizes that it is beneficial to react on changing computational demands during runtime. On the other hand, generic approaches allow introducing adaptivity straightforwardly to new application scenarios. Before we develop an adaptive simulator that combines both aspects in Chapter 4, the next chapter introduces established categorization

approaches for adaptive software, connects these to the presented adaptive simulators, and presents techniques needed to develop complex adaptive software.

# Chapter 3

# Adaptive Software

> We may ask ourselves whether the execution of a simple branching statement, like `if tooHeavy then askForHelp else push` can be interpreted as a form of adaptation. The answer is: it depends.
>
> Bruni et al. [20]

Typically, software that is able to change its behavior according to input and environmental changes is called adaptive software, see Definition 7, page 12. The need of adaptive software in general is usually motivated by two arguments. First, the ever increasing complexity of software that requires a software to autonomously manage itself by adapting to various changes [92, 105, 135]. Second, the ever increasing agility of software development making it difficult to define a solid software specification so that adaptability is needed to react on changed requirements or unforeseen changes [120, 138]. Although the given definition of adaptive software is vague and abstract, it is used frequently, e.g., [66, 120, 147, 178]. Nevertheless, based on this definition, almost any software can be defined as adaptive software depending on the point of view, e.g., Bruni et al. state that "[...] the judgment whether a system is adaptive or not is often subjective"[20]. However, several categorization approaches and classification concepts have been developed so far to structure the analysis of adaptive software.

Figure 3.1 illustrates a spectrum of adaptivity developed by Oreizy et al. in 1999 [147]. Referring to this spectrum, adaptive software that uses simple conditional expressions to change its behavior is least adaptive. Online algorithms that use the history of inputs to change their behavior are more adaptive. Generic or parameterized algorithms have parameterized behavior that can be changed based on inputs, e.g., by generic type instantiation. The algorithm selection category refers to adaptive algorithms that use properties of the environment to switch to the most suitable algorithm among a predefined set of options. Finally, the most adaptive software is

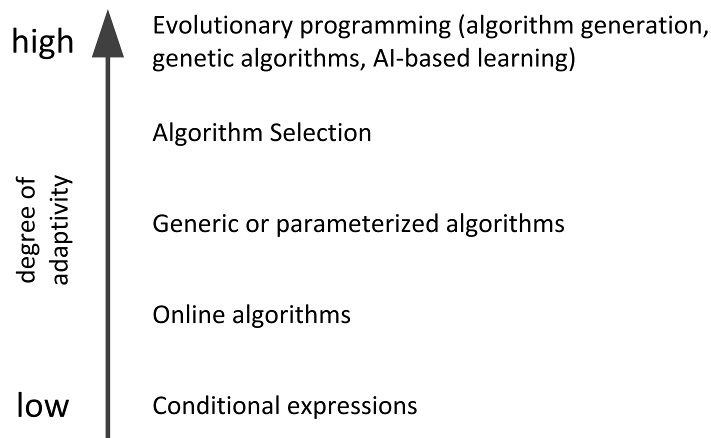



Figure 3.1: The spectrum of adaptivity from low (bottom) to high (top) based on [147].

able to create its adaptation options on its own by using some kind of evolutionary programming.

This simple spectrum seems to be useful at first sight as it supports the intuition that some approaches for adaptivity like conditional expressions are less adaptive than more complex approaches like evolutionary programming. Apparently, it can also be applied to adaptive simulators presented in Chapter 2. Some methods simply increase or decrease a primitive parameter after every simulation step execution depending on a predefined conditional expression, e.g., the *Adaptive Blocking Window* [8] or *Adaptive Throttling* [115]. Others exchange the simulators during runtime and perform some kind of algorithm selection, e.g., *LSODA* [158]. However, *LSODA* also uses a simple conditional expression to realize the decision-making process. Compared to the selection framework SASF [49], *LSODA* seems less adaptive as it cannot deal with a set of available simulators, but it only chooses between two pre-selected options (one simulator suited to stiff systems, one simulator suited to non-stiff systems). These examples emphasize that the spectrum of adaptivity shown in Figure 3.1 neglects many properties of adaptive algorithms that are necessary for a comprehensive categorization. Nevertheless, this spectrum underlines an important facet of adaptive software from rather simple to sophisticated variants that is commonly supported [135, 170, 30, 4].

The shown spectrum of adaptivity emphasizes that it is necessary to identify more properties of adaptive software for an effective analysis and categorization. A more comprehensive discussion about properties of adaptive software is given by Andersson et al. [3]. Four groups of facets are identified:

1. **Goals** the software under consideration should achieve.

2. **Changes** that cause adaptations.

Table 3.1: Exemplary illustration how some of the adaptive simulators presented in Chapter 2 can be mapped to the four facets of adaptive software [3].

| Simulator / Framework / Technique | **Goals** | **Changes** | **Mechanisms** | **Effects** |
|---|---|---|---|---|
| *Step Doubling* and *Embedded Error Estimation* [159] | fixed and conflicting goals: reduce runtime and be accurate as possible | strong dynamic adaptations: adapt after each simulation step | parameter adaptation (step size), no user intervention | adapt accuracy to the acceptable threshold |
| *Switching Time Warp* [184] | fixed goal: reduce runtime | strong dynamic adaptations: adapt after a threshold for rollbacks is crossed (not foreseeable) | parameter adaptation (LP prioritization), no user intervention | reduce number of rollbacks |
| *Adaptive Sorted Reactions* [133] | fixed goal: reduce runtime | strong dynamic adaptations: adapt after each simulation step | parameter adaptation (reaction order), no user intervention | reduce costs of reaction selection |
| *Hybrid Simulator for Biochemical Networks(1) [71]* | fixed conflicting goals: reduce runtime and be accurate as possible | weak dynamic adaptations: split reactions (slow vs. fast) once during initialization | parameter adaptation (reaction partitioning), no user intervention | use a partitioning resulting in an acceptable loss of accuracy |
| *Unsupervised Simulator Selection [47]* | fixed goal: reduce runtime | weak dynamic adaptations: select simulator once during initialization | compositional adaptation (dynamic set of simulators), no user intervention | use an efficient simulator to execute a simulation run |

3. **Mechanisms** that refer to the adaptation process itself.

4. **Effects** of applied adaptations.

Table 3.1 illustrates exemplarily how some adaptive simulators presented in Chapter 2 can be mapped to these facets.

Important properties of the **goals** facet refer for example to the number of goals of the system, the relation between goals (e.g., conflicting goals), the rigidity of goals or whether goals are fixed or dynamic. The goal of most presented adaptive means used for simulators in Chapter 2 is to improve the runtime efficiency of a simulation run. Adaptive strategies used for simulators trading accuracy for speed, e.g., *step doubling* [159] and $\tau$-leaping [25], have to deal with conflicting goals, i.e., perform as fast as possible and be accurate as required. Few methods have other goals. For example, besides improving the runtime performance, the method developed by Child and Wilsey [31] also pursues reducing the energy consumption of the used processors.

With respect to **changes**, the source (e.g., environment, application, infrastructure), type (e.g., functional, non-functional, technological) and frequency of a change should be considered. Further, it should be determined whether a change is foreseeable or not. In addition, McKinley et al. also distinguish between adaptations applied before a program runs (static adaptations), i.e., during development time, compile time, and load time, and adaptations during runtime (dynamic adaptations) [135, 169]. Referring to adaptivity in the context of modeling and simulation, this categorization has to be extended, see Figure 3.2. We refine the category of dynamic adaptations to *weak* and *strong* dynamic adaptations. *Weak* dynamic adaptations are executed during runtime, but only once during the initialization of a simulation run. *Strong*

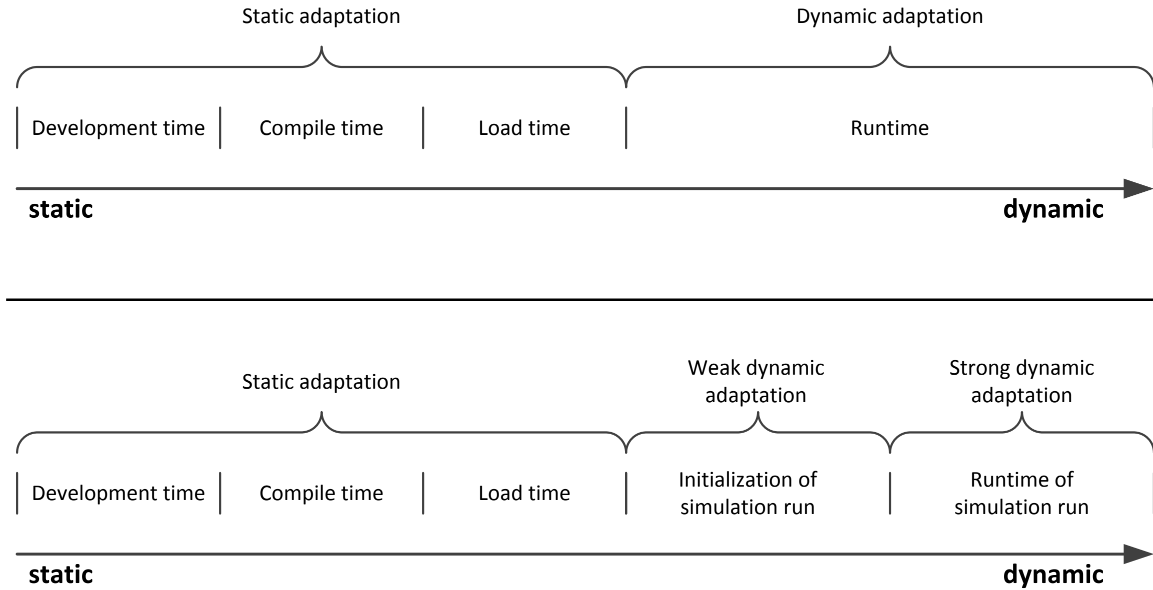



Figure 3.2: Top: Temporal classification of adaptivity from static (left) to dynamic (right) from [135]. Bottom: Extended temporal classification of adaptivity refining the dynamic adaptation into two categories: adaptations executed during the initialization of a simulation run (*weak dynamic adaptation*) and adaptations executed during a simulation run (*strong dynamic adaptation*).

dynamic adaptations are also executed during the execution of a simulation run. In general, software realizing dynamic adaptations is also referred to as *self-adaptive software* [3]. Most adaptive approaches presented in the previous chapter perform strong dynamic adaptations, see Table 2.1. Further, the adaptation frequency is mostly time-dependent, e.g., perform an adaptation every $n$ simulation steps, and rarely event-dependent.

An important property of the **mechanisms** facet is the type of adaptation, i.e., whether it is parametric or structural. Although using different terms, this distinction is made by various authors. For example, McKinley et al. introduce the terms *parameter adaptation* and *compositional adaptation* to describe this property [135]. Analogously, Salehie and Tahvildari introduce the terms *weak adaptation* and *strong adaptation* [170]. Independent of the concrete terms, they refer to similar concepts. Basically, a program that applies parameter or weak adaptation is able to modify and tune program variables. Moreover, simple forms of strategy selection are also assigned to this category, even if they exchange components of the software and hence change the structure of the software. It is only important that the strategies are contained in a fixed set of options that have been defined during the development of the software.

Thus, this kind of adaptation does not allow new options, algorithms or strategies to be added to the software after finishing its development. In this sense, the terms *compositional adaptation* and *structural adaptation* might be confusing, as the structure of a software can also be changed by strategy selection with *parameter adaptation*. In contrast, structural or compositional or strong adaptation refers to changing, adding, removing or substituting algorithmic or structural system components from a *dynamic* set of options, i.e., it allows adding new algorithms and options during runtime. Consequently, this type of adaptation is more flexible, however, it is also more difficult to implement, as a dynamic set of options must be considered. Further, it is more difficult to ensure the correctness of the software and possibly malicious components have to be identified. To illustrate the difference of both adaptation types, Salehie and Tahvildari give several examples in [170]. Other properties of the **mechanisms** facet deal for example with the autonomy of adaptations, i.e., whether humans are involved and the duration of adaptations. Most adaptive approaches presented in Chapter 2 clearly refer to parameter adaptation as they only change primitive or complex parameters. Further, the selection approaches LSODA [158], *ODExpert* [102], *ModSimPack* [21], and the methods developed in [33, 32] also only perform parameter adaptations, as they deal with a fixed set of options that cannot be changed. In contrast, the SASF [49] performs compositional adaptations, as it is able to select simulators from a arbitrary set of options that can change during runtime. Further, machine learning is used to regularly evaluate the available options. Referring to autonomy, *ODExpert* interacts with the user in case it cannot automatically calculate a property of a system, e.g., its stiffness. All the other approaches do not require user-intervention.

Finally, properties of the **effects** facet describe the risk of adaptations (What happens in case an adaptation fails?), whether adaptations are deterministic or non-deterministic or the benefits of adaptations. None of the presented adaptive approaches for simulators deal explicitly with failing adaptations, e.g., what to do if the decision-making process crashes. However, the SASF has a mechanism to deal with failing simulators, i.e., if a chosen simulator crashes, it is added to a blacklist and the simulation run is repeated with another simulator.

Besides the discussed low-level categorization concepts of adaptive software, a more high-level approach is to consider the $self-*$ properties [105, 140]. The four most prominent properties are a) **self-configuration**, i.e., the software must be able to configure itself depending on a high-level policy, b) **self-optimization**, i.e., the software continually tries to improve its performance, c) **self-healing**, i.e., the software is able to automatically detect and solve software and hardware problems, and d) **self-protection**, i.e., the software is able to identify and protect itself against malicious

attacks. All adaptive means presented for simulators refer to **self-configuration** and **self-optimization**.

Altogether, no adaptive simulator presented in Chapter 2 executes strong dynamic adaptations and is also generic like the methods of the SASF, i.e., performs compositional adaptations. Based on the explained facets of adaptive software, we can classify such a generic adaptive simulator as follows. The goal of this simulator would be to improve the runtime performance of a simulation run. Further, adaptations have to be executed during the execution of a simulation run, i.e., strong dynamic adaptations have to be executed. However, the trigger of adaptations is not obvious, e.g., whether to use a time-based or an event-based trigger. To be generic like the SASF, the adaptive algorithm must be able to deal with an arbitrary set of adaptation options and therefore must perform compositional adaptations and it must use machine learning to automatically evaluate the available options. Therefore, in the next Section 3.1 we firstly deal with main technological characteristics needed to realize compositional adaptations. Afterward, in Section 3.2, we discuss concrete techniques to implement adaptive software realizing compositional adaptations.

## 3.1 Concepts for Compositional Adaptations

McKinley et al. define three key technologies that must be supported by a software to enable compositional adaptations: a) **separation of concerns**, b) **component-based design**, and c) **computational reflection**. Separation of concerns is an old concept in software engineering that emerged around 20 years ago [95]. This paradigm enhances the even older concept of modularization in software engineering [150]. The idea of modularization is to develop a separated program module for each task of a software to improve its flexibility and comprehensibility and additionally to reduce its development time. Parnas already determined in 1972 that each module of a software should refer to a specific feature and not to a specific step of a software [150]. Basically, this is the idea of separation of concerns: each concern of a software should be separated from all the other concerns. Concerns refer to a variety of software properties like its business logic, concurrency, real-time constraints, logging, security issues or failure recovery. The motivation is similar to the motivation to modularize a software, i.e., it should be easier to write, understand, reuse and modify. However, a problem with separation of concerns is that there is usually no exact idea of what a concern is or what separation eventually means [46].

Referring to separation, Hürsch and Lopes distinguish the conceptual level and the implementation level of a software and that software must separate concerns on both levels [95]. At the conceptual level, clear definitions and conceptual identifications

of concerns are needed. However, it is not enough to identify the concerns and separate them conceptually, they must also be separated at the implementation level, i.e., each identified concern must be adequately isolated so that the code of several concerns is not intertwined. A separation of concerns that takes not place at the implementation level would result in code that is hard to understand, to maintain and to modify. In object-oriented languages, concerns can be separated by using class architectures. Nevertheless, it is not possible to clearly separate crosscutting concerns [106], e.g., logging and security capabilities. To separate such crosscutting concerns suitably, aspect-oriented programming can be used that allows implementing aspects (i.e., concerns) of a software separately (see Section 3.2). Besides, architectural approaches exist to apply separation of concerns, e.g., design pattern like the Model-View-Controller [111] or service-oriented architectures [45].

Following the separation of concerns paradigm, a software design typically also follows a component-based architecture in which each concern is encapsulated by a loosely coupled component. Referring to adaptive software, this architecture is well suited as it allows composing a software from a set of components and supports the implementation of recomposition features to change the composition during runtime. To implement a component-based architecture, standardized specifications exist, e.g., for Java the module framework OSGi (Open Service Gate Initiative) [2]. In OSGi, components are called bundles and a runtime infrastructure is specified to add, replace and remove bundles. A key aspect of OSGi is a service registry that manages all registered bundles and provides means for the bundles to publish their services and to retrieve published services from other bundles [187]. A similar concept is realized within the modeling and simulation framework JAMES II [87]. Here, a component is called a plugin and a central registry is used to maintain the set of active plugins that can be changed during runtime.

Besides separation of concerns and a component-based design, McKinley et al. propose the implementation of computational reflection. Computational reflection is defined by Maes as “the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation” [125]. To be reflective, a software must be able to monitor itself to collect various information like performance statistics or function calls. This feature is often referred to as *monitoring* [177] or *introspection* [135]. Moreover, a reflective software needs capabilities to change its behavior, e.g., by reconfiguration or recomposition. This feature is often referred to as *reconfiguration* [177] or *intercession* [135]. Typically, a reflective software consists a) of a base level that contains objects and their connections and b) of a meta level containing meta objects that encapsulate information about objects of the base level. Every base level object is connected with exactly one corresponding meta

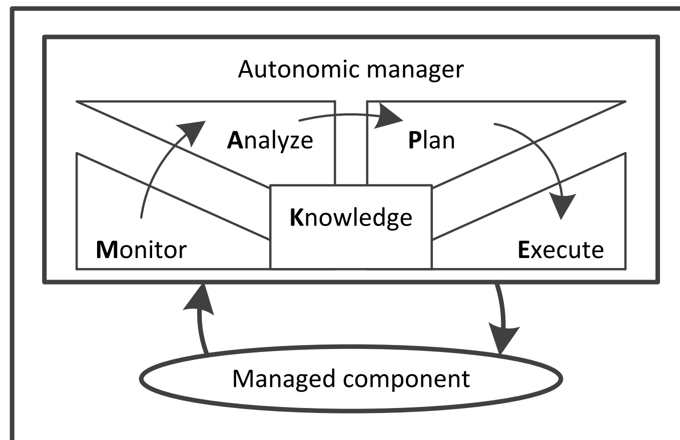



Figure 3.3: The MAPE-K control loop based on [105].

object. Both levels are causally connected so that a change in one level affects a change in the other level, e.g., removing an object also removes its meta object. Available modifications and changes are specified and executed via metaobject protocols. These protocols guarantee consistent states of the base level and meta level. Reflective software can either be realized by explicitly implementing a reflective architecture [22, pp.193-219], or by using native reflection functions of programming languages. For example, the reflection API in Java allows to perform introspection by accessing meta information about objects like their class name and available methods during runtime.

Besides the discussed three characteristics, da Silva also mentions **decision making** as a major requirement for adaptive software [177], i.e., the ability to use observed information to decide which adaptations to execute. For decision making, static conditional expressions can be used. In this case, the developer of the adaptive software must be aware of a complete mapping of all possible observations to available actions. Thus, the set of actions as well as the set of possible observations must be known during development time and they cannot change during runtime. More flexibility can be achieved by using machine learning techniques for the decision making [34, 15, 178].Typically, these techniques are able to learn a suitable mapping themselves and can therefore deal with a changing set of actions and a changing set of observations.

Moreover, sophisticated adaptive methods should be implemented separately from the business logic of the software by using a control loop design [30]. A well-known control loop usually referred to as the *MAPE-K* control loop has been developed by Kephart et al. in 2003 [105], see Figure 3.3. Here, four basic activities are distinguished: **monitor** (observing the environment and itself), **analyze** (analyze observed information), **plan** (construct an action plan based on the analysis), and

**execute** (execute the constructed plan). Further, a common knowledge component shall be used by all activities. A similar control loop has been developed by Dobson et al. in 2006 that describes the same activities but with different terms: collect (instead of monitor), analyze, decide (instead of plan), and act (instead of execute) [38].

The SASF, as the only adaptive approach presented in Chapter 2 that performs compositional adaptations, follows these technological characteristics. It separates all important concerns like the decision-making process, the performance monitoring, or the data handling into components realized as JAMES II plugins. Further, by exploiting the plugin system of JAMES II, it is able to reflect on the structure of simulators. Further, it observes the performance of the selected simulators and applies reinforcement learning to use gained data to evaluate the available options.

The presented characteristics refer to conceptual strategies to realize sophisticated adaptive software. The following section presents various concrete techniques to implement these concepts.

## 3.2 Techniques to Implement Adaptive Software

An adaptation ability can be added to software by using various software engineering techniques that usually base on some kind of indirection [135], e.g., applying inheritance as done by the `AdaptiveSimulationRunner` (see Section 2.5.3) or using the *wrapper*[1] pattern [60]. The wrapper pattern can be used to extend the functionality of an object dynamically by encapsulating it with other objects, see Figure 3.4. Using this pattern, a wrapper realizing the adaptivity control loop can control its wrapped object and modify or replace it if necessary. An advantage of the wrapper pattern is that the wrapper implements the same interface as the component, i.e., an object that is using a component does not have to be changed to use the wrapper. Another approach especially in distributed systems to implement the adaptivity control loop is to use adaptive middleware concepts, e.g., the mobility- and adaptation-enabling middleware MADAM [57]. Here, the middleware observes the performance of existing components and modifies, replaces or deletes them as necessary. A comprehensive survey of various adaptive middleware architectures and frameworks is given by Sadjada [168].

A prominent programming paradigm that is suitable to add adaptivity to a software is aspect-oriented programming (AOP) [106]. The motivation of AOP is to enable the separated implementation of crosscutting concerns that cannot be encapsulated neither with procedural nor with object-oriented programming techniques, i.e., crosscutting concerns do not fit the modularization of the software [89]. By separately implementing crosscutting concerns in aspects, the comprehensibility and maintainability of software

[1] Also known as the *decorator* pattern.

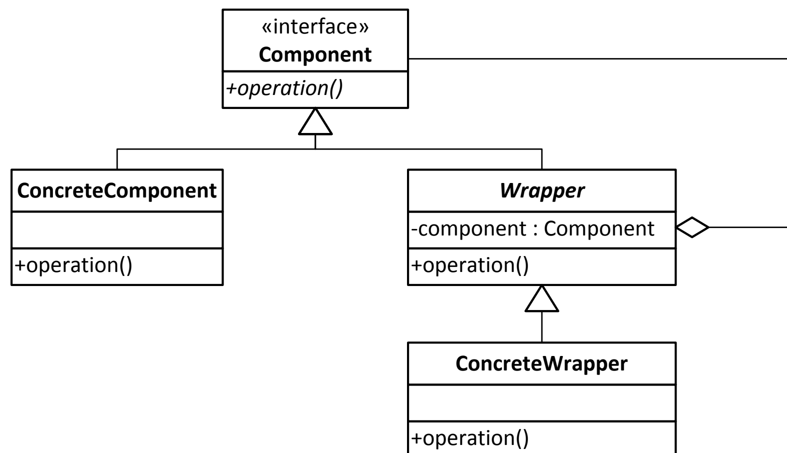



Figure 3.4: UML class diagram of the wrapper pattern [60].

can be improved. For example, Lippert and Lopes achieved a significant reduction of complexity by using aspect-oriented programming [122]. Mainly, redundant condition checks and exception handling code have been aggregated. During the compilation process of an aspect-oriented program, an aspect weaver intertwines all aspects with the base code to a working software. For this, join points must be specified that are used to add aspect code into the program. Basically, a join point can be any point in the program flow, e.g., when an arbitrary method of a specific class is called. Referring to adaptive software, approaches have been developed that allow weaving aspect code during runtime with the base code, e.g., CaesarJ [5]. Although a promising programming paradigm, AOP can complicate the software development. For example, not all aspects of a software are typically identified at the beginning of the development phase and subsequent structural changes are tedious to apply [139]. Further, case studies have shown that AOP is only beneficial if aspects are updated frequently [43].

Another programming paradigm that can be used for the developing of adaptive software is context-oriented programming (COP) [89]. The purpose of COP is to enable software entities to adapt their behavior during runtime based on the current execution context. Adaptation to the current context is often an aspect that crosscuts the application logic (orthogonal to the modularization) [172]. For this, contexts that can contain any information that is computationally accessible are treated explicitly. The context can either be defined globally for the whole software of locally for each software component. Based on the current context, functions can for example be modified, extended, activated or deactivated. However, existing approaches typically require a static context handling, i.e., for each context state, the implications must be implemented explicitly, e.g., [4].

## 3.3 Summary

Several approaches exist to categorize adaptive software. Unfortunately, the term "adaptive software" is often used ambiguously and no unique and accepted definition exists. The designed facets developed by Anderson (**goals**, **changes**, **mechanisms**, **effects**) suitably aggregate many ideas [3] and should be considered when adaptive software is developed, analyzed, and evaluated. However, the developer of the adaptive methods used for simulators presented in Chapter 2 hardly consider such facets explicitly to analyze their methods. The goal of these methods is to improve the runtime performance of simulation runs, whereby approximate methods must also deal with the conflicting goal to achieve a suitable accuracy. Most presented methods perform strong dynamic adaptations, i.e., they apply adaptations during the execution of a simulation run. Further, time-triggered as well as event-triggered adaptations are common. All methods but the *Supervised Simulator Selection* and the *Unsupervised Simulator Selection* perform parameter adaptations. Nevertheless, these two methods only perform weak dynamic adaptations, i.e., they select a simulator during the initialization of a simulation run. As written in Section 2.6, combining strong dynamic adaptations and compositional adaptations is a promising approach we pursue in Chapter 4.

To realize compositional adaptations — from a technological viewpoint — separation of concerns, component-based design, and computational reflection are key characteristics of adaptive software realizing compositional adaptation [135]. Further, the decision-making process plays an essential role and should not be intertwined with the business logic of a program. Various concrete techniques exist to implement adaptive software, e.g., software pattern, aspect-oriented programming and middleware architectures.

Based on the conceptual requirements for compositional adaptations, the modeling and simulation framework JAMES II is a suitable simulation software to be used as a basis for an adaptive simulator combining strong dynamic adaptations with compositional adaptations. Separation of concerns has always played a central role in the development of JAMES II [87], e.g., it follows a strict separation of model and simulator and it has an explicit experimentation layer [48]. The plugin system ensures a component-based design and allows reflecting about the structure of algorithms, simulators and data structures.

Based on the analysis of various adaptive means for simulators in Chapter 2 and the discussed properties, concepts and techniques for adaptive software in this chapter, a generic and dynamic adaptive scheme can now be designed and implemented, see Chapter 4.

# Chapter 4

# The Adaptive Simulator — Compositional Simulator Adaptation at Runtime

> The key to solving computationally challenging problems lies in a combination of design choices, with effects on performance often interacting in complex, unexpected ways.
>
> Holger H. Hoos [91]

Adaptivity is broadly used in modeling and simulation, especially to improve the runtime of a simulation run, see Chapter 2. Its usage is motivated by changing computational demands during a simulation run. These changes can be imposed by the model: for example, a changing number of model entities, a different kind of event to be processed, or structural changes within the model. Further, similar effects can be caused by changes in the execution environment, such as a change in the number of available processors, CPU load from other jobs, or changes in network latency.

However, as shown in Table 2.1 (p. 31), many adaptation mechanisms in modeling and simulation that perform *strong dynamic adaptations* are tailored to an application scenario and merely perform *parameter adaptations*, see Chapter 3. Developing generic mechanisms realizing *compositional adaptations* is challenging. This is emphasized by the work of Ewald et al. that focuses on the selection of a simulator during the initialization of simulation runs, see Section 2.5. The effectiveness of the approach using supervised learning (Section 2.5.2) essentially depends on collected past performance data and suitably selected problem features. While the first problem can be solved partly by using benchmark models, the second problem is domain-dependent and cannot automatically be solved completely. In contrast, the unsupervised selection

approach (Section 2.5.3) does not depend on past performance data and problem features: it learns which algorithm to choose on demand by using multi-armed bandit policies. Nevertheless, this approach requires many replications only useful in case of stochastic simulation runs. Although both approaches are rather generic and have shown the potential to be effective, they do not perform *strong dynamic adaptations*.

This chapter introduces a generic adaptive method realizing *strong dynamic adaptations* and *compositional adaptations* applicable to any simulator. This method is neither restricted to a specific simulator, nor is it restricted to adjusting algorithm parameters. Due to this generality, we call this method simply the `Adaptive Simulator`.

## 4.1 Requirements

The purpose of the `Adaptive Simulator` is to improve the performance of a simulation run execution by adaptations during this execution. Obviously, it must be therefore possible to apply adaptations during the execution of a simulation run. However, it should be sufficient to allow adaptations between the execution of two simulation events, as the state of the model and simulator may not be well-defined during the execution of an event. Also, none of the adaptive approaches considered in Chapter 2 apply adaptations *during* a step execution.

Further, the `Adaptive Simulator` must be able to estimate its current performance. Consequently, a well-defined performance metric must be provided. The metric must consider all performance aspects that can be changed by adaptations and that are of importance for the user. For example, if the runtime of the simulation run and the accuracy of the results are important for the user and both can be influenced by adaptations, then both properties must be considered for the performance calculation. If only the runtime would be considered, selected adaptations would probably result in inaccurate simulators that are fast but not sufficiently accurate. Besides, the performance calculation itself must be simple. For example, evaluating the runtime performance could be done by calculating the event throughput. However, measuring the accuracy of simulation results is typically not done directly based on simulation results, but a specific level of accuracy is implicitly guaranteed due to simulator configurations, e.g., see $\tau$-leaping [25]. Explicitly estimating the accuracy of simulation results is complex and thus might not be suitable to be used as a performance metric.

To be applicable generically and with little user intervention, a) the `Adaptive Simulator` must be able to compute the set of its adaptation options automatically with the help of the simulation system, b) it should use a learning algorithm to automatically learn when to apply which adaptation, c) it should identify suitable

adaptation trigger automatically, and d) it should not require past performance data or previous analysis. Using learning techniques for the `Adaptive Simulator` makes it reusable, however, these techniques come with own challenges that must be solved. Referring to the `Adaptive Simulator`, it is especially difficult to solve these challenges due to the wide variety of modeling languages and models that they shall be used for. Consequently, solutions have to be robust for most scenarios and default configurations must usually provide sufficiently good results.

In general, besides the possible performance benefits of runtime adaptation, the `Adaptive Simulator` should also reduce the user's configuration effort. Selecting suitable configurations for concrete simulators is often difficult, however, it is probably even more difficult to configure a generic adaptive simulator that has too many "cryptic" meta-level parameters. Thus, the number of such parameters should be as low as possible and the effect of all existing parameters should be understandable.

Adaptations will probably change concrete simulation results especially in case of stochastic simulations. This can easily happen and is no problem, e.g., if a new random number generator is initialized during an adaptation. Statistically, simulation results would be still correct after such changes. Nevertheless, the `Adaptive Simulator` must not be allowed to actively take control over the model evolution, e.g., avoid the occurrence of rare events that are complex to be calculated. Further, the adaptation process must guarantee the integrity of the simulator, e.g., all data structures have to be updated properly — a key challenge for adaptive software [135].

Finally, it must be possible for several `Adaptive Simulator` instances to use the same knowledge base. By using multicore machines, several simulation runs are typically executed in parallel, e.g., replications or simulation runs with different configurations of the same model. However, even when simulation runs are executed sequentially, the `Adaptive Simulator` should not be initialized with an empty knowledge base and learn from scratch but it should use an existing knowledge base when there is a suitable one available.

## 4.2 The Structure of the Adaptive Simulator

In JAMES II, every simulator inherits from the class `Processor`[1] that has the important method `nextStep()`. This method is used to execute an atomic step of a simulation run, e.g., execute the next event. Consequently, *all* simulators in JAMES II

[1]Initially, the `Processor` class has been named `Simulator`, but the developer of JAMES II thought that this name is too restrictive, as not only simulators, but all types of entities that process some data can be implemented by using this class. However, here, it will be used only in the context of simulators.

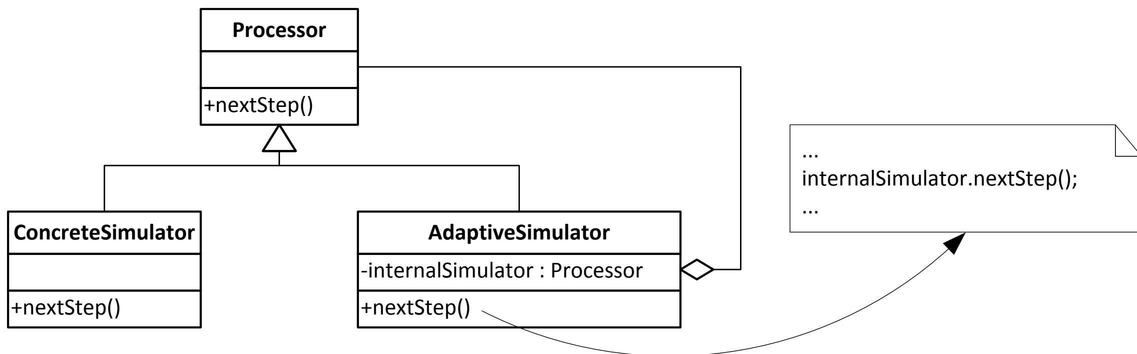



Figure 4.1: The wrapper pattern used by the `Adaptive Simulator`. During the `nextStep()` method, the `Adaptive Simulator` performs analysis tasks and reconfigure or exchange its `internalSimulator`.

inherit from the same class and use the same method `nextStep()` to proceed with the simulation. For the `Adaptive Simulator` that should not depend on a concrete simulator, nor on a specific modeling language, it is in this case reasonable to exploit this architecture and to apply the wrapper pattern (see Section 3.2, page 42) as illustrated in Figure 4.1. As every simulator, the `Adaptive Simulator` inherits from the class `Processor`. However, it is not able to execute any simulation step *directly*, but it uses an internal simulator to execute the actual simulation run. Further, by overriding the `nextStep()` method, the `Adaptive Simulator` can perform analysis before and after the step execution with its internal simulator and moreover it can reconfigure or exchange this internal simulator. Generally speaking, the control loop (see Section 3.2) realizing the adaptive behavior can be realized there. Thus, adaptations can be executed during the execution of a simulation run between the execution of individual simulation events. Following this idea means that the `Adaptive Simulator` can be used as every other simulator, i.e., the adaptation process is integrated transparently. Nevertheless, it also means that adaptations cannot be performed *during* the execution of a simulation step — a consequence that on the other hand also simplifies the adaptation process, e.g., because the state of the model is well-defined before and after the execution of a simulation step, but not necessarily *during* its execution. Integrating the `Adaptive Simulator` into JAMES II also allows calculating all valid simulators and simulator configurations, i.e., the action set $A$, automatically like done by the SASF, see Section 2.5.

For the decision-making process, i.e., how to adapt the internal simulator during a simulation run, the machine learning technique reinforcement learning [185] is suit-

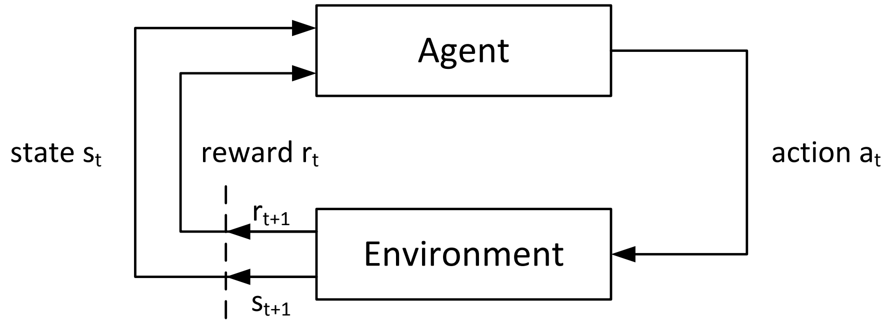



Figure 4.2: The basic model of interaction in reinforcement learning based on Sutton [185].

able [178]. Reinforcement learning works incrementally and does not require training data. First, an agent observes a state $s \in S$ of a (non-deterministic) environment and receives a numerical reward for its previous action, see Figure 4.2. Next, it decides which action $a \in A$ to execute for achieving some goal, e.g., to reach a specific state of the environment. Formally, this is a Markov decision process [13]. Typically, the goal is encoded in the received numerical rewards. The whole procedure (observe, analyze, select, execute) is repeated afterward, i.e., the new state of the environment is perceived and the agent has to decide upon its next action. Eventually, the agent's task is to maximize the overall received reward. It has typically neither prior knowledge on the actions' effects, nor on the desirability of certain environment states (in terms of rewards), nor on the transition probabilities between states. Thus, the agent has to *explore* the actions and the states of its environment, but it also has to *exploit* its (incrementally growing) knowledge to increase its reward, so that eventually the best action is chosen in any given situation. Moreover, delayed rewards have to be considered, i.e., in the long run actions with low reward might result to situations with potentially high rewards. Altogether, reinforcement learning is nicely compatible with a control loop design for adaptive software like *MAPE-K* (see Section 3.1), i.e., the available actions of the agent correspond to the available adaptations and the states of the environment correspond to the data the adaptive component can monitor.

A popular reinforcement learning algorithm is Q-Learning [196]. Q-learning is a temporal-difference learning method, i.e., it estimates the utility of a state $s$ by also considering the estimated utility of the successor state $s'$ and it does not require any prior knowledge on the environment's state transitions. Q-learning learns how valuable it is to take an action $a \in A$ after observing a state $s \in S$ in form of q-values: the higher the q-value for a state and an action, the better it is to take this action after observing this state. All q-values can be represented by an $|S| \times |A|$ matrix $(Q(s, a))$. In this simple case, neither an infinite number of states nor an infinite number of

actions is allowed. The learning rule of Q-learning is defined as follows:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \cdot [r_{t+1} + \gamma \cdot \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{4.1}$$

The q-value $Q(s_t, a_t)$ is updated by using the current q-value, the received reward $r_{t+1}$ and the best possible q-value of the successor state $s_{t+1}$, i.e., $max_{a \in A} Q(s_{t+1}, a)$. The best possible q-value of the successor state is multiplied by a discount factor $\gamma \in [0, 1]$. The discount factor can be used to configure the influence of future rewards on the current q-value. Further, a learning rate $\alpha \in [0, 1]$ is used to configure the learning speed. It usually decreases for $a_t$ and $s_t$ the more often $a_t$ has been chosen after observing $s_t$. The action selection in Q-learning can be done by multi-armed bandit policies [7].

Abstracting from the concrete realization within the experimentation layer in JAMES II, the basic algorithm as realized by the `Adaptive Simulator` is described in Algorithm 4.1. The algorithm's outer loop (l. 6–33) covers the adaptive execution of a single simulation run, while the algorithm's inner loop (l. 11–15) covers the simulation between two adaptations. After the execution of a simulation event (l. 12), data from the model (e.g., the values of variables, the number of components, or the coupling structure), data from the internal simulator (e.g., the number of triggered events, the event queue length, or the usage of auxiliary data structures), and data from the environment (e.g., the number of cores available or the memory load) are collected (l. 13). One tuple of all collected data is referred to as a **base state** $\sigma \in \Sigma$. Base states represent the current phase of the simulation run [142], i.e., the features that determine the computational characteristics of the simulation execution. Consequently, the feature selection problem is not solved by the `Adaptive Simulator` directly, but by the developers of every component that is used.

Figure 4.3 illustrates the components of the `Adaptive Simulator`. Basically, the knowledge base saves the q-value matrix updated by Q-learning. The action selection uses the knowledge base to decide which action to choose during an adaptation. The selected action influences the internal simulator and this action must also be forwarded to Q-learning for the next learning iteration. The state handling deals with the observed data to calculate states for the reward function, Q-learning, and the adaptation conditions. The following paragraphs discuss these components in more detail.

**Adaptation Condition.** It is not reasonable to execute adaptations between every event execution per default, since a) probably the computational characteristics will not change that frequently and b) the decision-making process as well as the adaptation

**Algorithm 4.1** Pseudo-code for the `Adaptive Simulator`.
$Q$: q-value matrix indexed by aggregated state $s \in S$ and action $a \in A$.
$N$: matrix of counters for visited $(s, a)$ tuples.
$s, s' \in S$: previous and current aggregated state.
$a \in A$: action.
$r \in \mathbb{R}$: reward.
$\sigma \in \Sigma$: current base state.
$\tau \in \Sigma^*$: current base state trajectory (seq. of base states).
`internalSimulator`: current internal simulator.

```
1  s := s_0
2  a := a_0
3  internalSimulator = initialize(a)
4
5  // Adaptation loop
6  repeat {
7    N[s,a] := N[s,a] + 1
8    τ := []
9
10   // Simulation loop
11   repeat {
12     internalSimulator.nextStep()
13     σ := observe()
14     τ := τ + σ
15   } until adaptationCondition(τ)
16
17   // Calculate reward
18   r := R(τ)
19
20   // Process base state trajectory to state
21   s' := p(τ)
22
23   // Update knowledge base with Q-Learning
24   Q[s,a] := Q[s,a] + α(N[s,a]) · (r + γ · max_{a'∈A} Q(s',a') − Q[s,a])
25
26   s := s'
27
28   // Select next action based on s,Q,N
29   a := f(s,Q,N)
30
31   // Adapt internal simulator
32   internalSimulator := adapt(internalSimulator, a)
33 } until isTerminal()
```

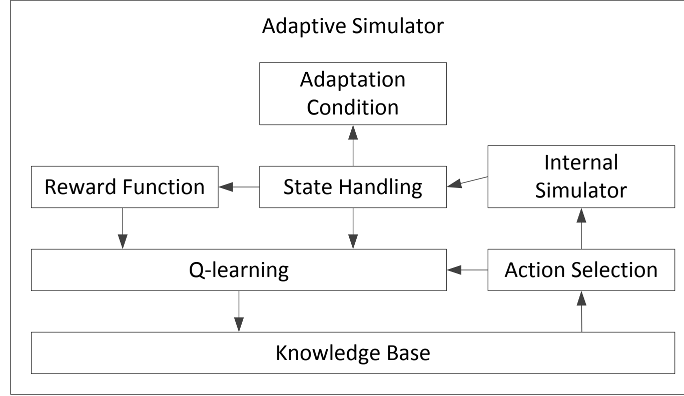



Figure 4.3: Overview and influences of the main components of the `Adaptive Simulator`.

execution cause additional computational costs. Further, algorithms with warm-up phases would not be considered suitably as they would not get the chance to exploit their full potential. An `adaptationCondition(`$\tau$`)` method is used to decide whether to perform an adaptation (l.15) and therefore a sequence of base states is available to represent the current phase of the simulation[2]. A sequence of base states defines a **base state trajectory** $\tau \in \Sigma^*$:

$$\tau = \sigma_1 \sigma_2 \ldots \sigma_n.$$

Various possibilities exist to realize the `adaptationCondition(`$\tau$`)` method, e.g., it could be static, so that an adaptation is executed after a fixed number of simulation events, or it could be more flexible, so that an adaptation is executed depending on properties of $\tau$. Section 4.4 presents several possibilities in more detail.

**State Handling.** Base states can be high-dimensional and a base state trajectory can include many base states. Using a base state trajectory directly for a learning algorithm will therefore probably result in the *curse of dimensionality* [14], i.e., the number of states is too high for a suitable learning effectiveness. Consequently, after the reward for the current base state trajectory $\tau$ has been calculated, it is processed into an **aggregated state** $s' \in S$ by using a function $p : \Sigma^* \to S$ (l. 21). Aggregated states represent condensed base state trajectories. They have fewer dimensions and should therefore be more suitable for the learning algorithm.

A useful approach to realize the function $p$ is to divide it into two processes,

[2]In contrast to our initial approach in [75] and [76], the adaptation conditions are not integrated into the actions.

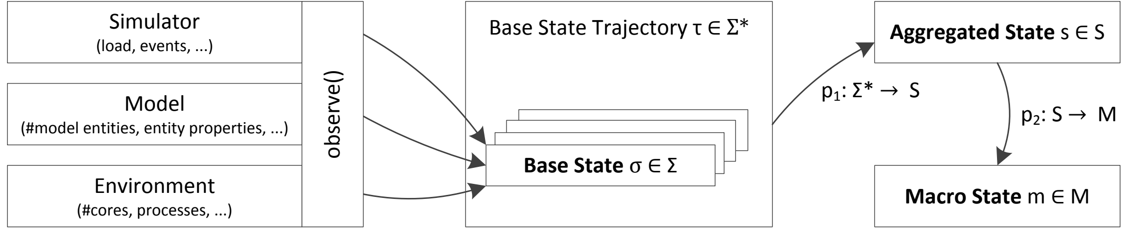



Figure 4.4: Overview of the state observation, aggregation of base state trajectories and generalization of states of the `Adaptive Simulator`.

$p_1 : \Sigma^* \to S$ and $p_2 : S \to M, M \subseteq S$, see Figure 4.4. The function $p_1$ is responsible to aggregate a sequence of base states to an aggregated state, e.g., by calculating the average values of all dimensions resulting in an "averaged" base state. Further, this function has to be able to deal with base states with different quantities of data. This is the case, for example, if an agent-based model is simulated, the number of agents changes during runtime and the age of each agent is used as one element of the base state tuple. Clearly, in such a case, a generic version of the function $p_1$ cannot be used, but a domain-specific variant has to be implemented that can exploit specific features of possible base states. For example, referring to the agent-based model by calculating the average age of all agents.

The function $p_2$ can realize a generalization of the calculated aggregated state. It partitions the space $S$ of the aggregated states into disjunct regions and maps all aggregated states of a region to the same **macro state** $m \in M$ that is used to represent the region, i.e., $\forall s \in S\ \exists! m \in M : m = p_2(s)$. Since $M \subseteq S$, a macro state $m \in M$ is always also an aggregated state, i.e., $m \in S$. Thus, Algorithm 4.1 does not deal with macro states explicitly. The advantage of the distinction between $p_1$ and $p_2$ is that only $p_1$ has to deal with a dynamic number of base states considering application-dependent knowledge. Further, $p_2$ can apply generic generalization methods for reinforcement learning and can be implemented application-independently. Section 4.3 discusses in more detail the opportunities and challenges of generalization within the `Adaptive Simulator`.

**Reward.** When an adaptation shall be executed, i.e., `adaptationCondition(`$\tau$`)` returns `true`, at first a reward $r \in \mathbb{R}$ is calculated by using the function $R : \Sigma^* \to \mathbb{R}$ (l. 18). The reward represents the performance of the current `internalSimulator` to execute the last $n$ simulation events resulted in $\tau = \sigma_1\sigma_2 \ldots \sigma_n$. The overall goal

of the `Adaptive Simulator` is to maximize the received rewards. As the runtime is often an important performance characteristic, the reward could simply represent the event throughput, i.e., the number of calculated events per second. In contrast to the `AdaptiveSimulationRunner` (see Section 2.5.3), using the runtime itself is not a suitable performance metric, as it does not consider the length of the simulation loop — a problem that is ignored in the context of the `AdaptiveSimulationRunner`: Only the runtime of a replication execution is measured without considering the executed number of simulation events. Besides, it can be advantageous to use a logarithmic event throughput as reward since it pronounces the differences between small event rates over those between high rates [74]. In principle, other metrics are also possible for the reward like the energy consumption or the accuracy of the simulation results. As written in Section 4.1, the used performance metric must at least reflect all characteristics that are of importance for the user and that can be changed by adaptations. Moreover, the impact on the reward of factors not considered in the base states should be small. This is often difficult to ensure even for runtime performance, since the concrete runtime can be influenced significantly for example by concurrently executed processes.

**Actions.** Besides the reward function $R : \Sigma^* \to \mathbb{R}$ and the state space $S$, the action set $A$ is central in the description of the `Adaptive Simulator`. Analogously to the `AdaptiveSimulationRunner`, selection trees can be used as actions (see Section 2.5.1), i.e., an action represents a complete hierarchical configuration of a simulation algorithm.

Executing an action $a \in A$, i.e., executing the method `adapt()`, means that the current `internalSimulator` of the `Adaptive Simulator` is changed so that the updated `internalSimulator` has the configuration represented by $a$ (l. 29). This procedure can be realized in different ways. Firstly, it could determine the differences between the old and the new action, and adapt only those points of the simulator (i.e., parameters, sub-plugins) that have to be changed. This procedure is useful in case only few elements like primitive parameter values have to be changed. Nevertheless, the updated `internalSimulator` must be valid, i.e., new plugins must be initialized properly. Further, the integrity and the validity of changed plugins have to be checked and guaranteed. To avoid these challenges, another approach for the `adapt()` method is to exchange the `internalSimulator` completely. The newly selected simulator is simply initialized with the current state of the model. The developers have therefore not to deal with complex initializations and integrity checks of plugins. Therefore, we implemented this generic `adapt()` method for the `Adaptive Simulator`. Nevertheless, the simulation system must provide model state objects that can be used to initialize the simulators — a requirement fulfilled in JAMES II due to the strict separation

between model and simulator.

**Learning Function.** After computing the aggregated state $s'$ based on the current state space trajectory $\tau$, the Q-Learning rule is applied to update the q-value of action $a$ and the previous aggregated state $s$ (l. 24). The learning rate $\alpha : \mathbb{N} \to [0, 1]$ depends on the number of selections already made for action $a$ after observing the aggregated state $s$. In the most simple case, $\alpha(N[s, a]) = \frac{1}{N[s,a]}$, i.e., the learning rate for action $a$ and the aggregated state $s$ converges to 0 with an increasing number of selections of this state-action pair. As usual, delayed rewards are considered discounted by the factor $\gamma \in [0, 1]$ by using the maximum q-value achievable for the current aggregated state $s'$. The purpose of delayed rewards is to consider long-term achievements, e.g., to justify low rewards that are necessary to reach a specific area within the state space that promises high rewards. Referring to the `Adaptive Simulator`, considering delayed rewards can be useful if simulators, data structures or sub algorithms have warm-up phases and are worth to be reused after an adaptation. For example, it might be worth to reuse a grid file [143] that is filled completely instead of creating a new empty grid file. To exploit this fact, the `adapt()` method would have to be realized in the way so that it is only changing the simulator as necessary and it is not exchanging the simulator completely.

The function $f : S \times \mathbb{R}^{|S|\times|A|} \times \mathbb{N}^{|S|\times|A|} \to A$ (l. 29) represents the policy responsible for action selection, i.e., a multi-armed bandit policy. Generally speaking, it determines the trade-off between exploration and exploitation. Analogously to the `AdaptiveSimulationRunner`, many policies can be used like $\epsilon$-greedy, $\epsilon$-decreasing, UCB1 [6], etc.

## 4.3 State Space Generalization

Due to high-dimensional and real-valued state spaces, it might be not feasible to learn selection policies for each state individually. Therefore, an agent typically generalizes the environment's states it perceives [185]. On the one hand, generalization reduces the learning effort due to a smaller number of distinguishable states; on the other hand, too much generalization may remove important distinctions in the state space, and thus reduces the potential for choosing the right action in the right state. To account for this trade-off, the degree of generalization must be chosen carefully. Basic reinforcement learning approaches use a predefined regular grid to partition the state space, so that the same degree of generalization is used for the whole space. This procedure is neither easy to use nor optimal. Firstly, the user has to configure the degree of generalization manually, which requires environment knowledge. Secondly, a

high degree of generalization might be suitable for specific areas of the state space, but unnecessary for other areas. Suppose an agent that tries to find the exit in a maze and its only information about the environment is its absolute position with millimeter accuracy inside the maze. Shall it generalize its position by rounding it to centimeters, decimeters, or meters? In a big, empty room, meters might be suitable. In front of a pit, centimeters might be a better choice. Dynamic aggregation algorithms partition the state space dynamically into disjunct macro states. This is necessary because it is not reasonable to manually determine a suitable partitioning.

Referring to the `Adaptive Simulator`, the size of the space $S$ of aggregated states depends on two parts:

1. The selected features of the model, simulator and environment to be observed.
2. The aggregation method $p_1$ that transforms a base state trajectory $\tau \in \Sigma^*$ to an aggregated state $s \in S$.

The first part depends on the developers of the modeling language, the simulation system and the plugins used by the simulators. Only information provided by the `observe()` method can be used by the `Adaptive Simulator` to distinguish simulation phases. Here, developer should already reasonably select information that could be of interest for the `Adaptive Simulator`. Nevertheless, deciding what information could be useful or not is challenging and in case of doubt, the information should be rather added. Thus, a base state $\sigma \in \Sigma$ will probably be high dimensional including continuous dimensions.

The aggregation of a base state trajectory $\tau \in \Sigma^*$ to an aggregated state $s \in S$ done by $p_1$ can be realized in various ways. For example, it can calculate averages of dimensions. It could also calculate minimum or maximum values of specific dimensions. Moreover, outlier base states could be emphasized or neglected. Further, one could analyze the performance of the applied simulator for each base state individually and decide, dependent on the performance, which base states are important and which are not. So far, we have not developed concepts for such sophisticated $p_1$ methods and only followed the idea of $p_1$ calculating the average of all base states — a simple approach assuming that the base states of one trajectory do not differ significantly and that the performance of a base state trajectory is determined averagely by all of its base states.

The function $p_2$ generalizes aggregated states to macro states. Consequently, $p_1$ is not allowed to be changed during runtime as the generalization would not be consistent in this case. The generalization creates a partitioning of disjunct regions within the state space by mapping all aggregated states of a region to the same macro state. In the most simple case, one static region can be used for the whole state space $S$, i.e.,

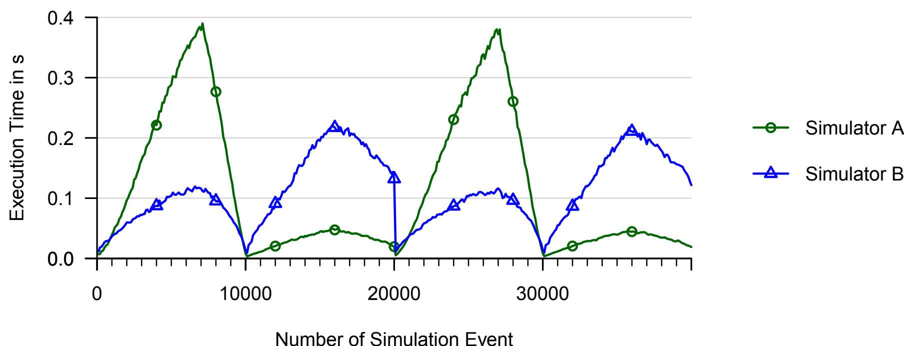



Figure 4.5: From [76]. Execution times for the two ML-Rules simulator configurations for the ML-Rules benchmark model. Each data point shows the execution time summed over 100 simulation events.

$\forall s \in S : p_2(s) = m$ so that $M = \{m\}$. Consequently, all observed features would be ignored and no simulation phases with different computational demands could be distinguished. However, this approach could also be useful, as it reduces the learning problem to a multi-armed bandit setting [6]. More interesting generalization methods can be implemented statically, i.e., by simply rounding continuous values or by using a regular grid over the state space. However, dynamic generalization algorithms can also be applied in case of metric dimensions. In Section 4.3.1, the **decision boundary partitioning algorithm** [165] is applied to the `Adaptive Simulator`. In Section 4.3.2, the **adaptive vector quantization algorithm** [114] is applied to the `Adaptive Simulator`. To illustrate the generalization concepts, a running example is given at the end of each section.

## Running Example Part 1

Suppose the `Adaptive Simulator` shall be applied to ML-Rules (a modeling language to simulate biochemical reaction networks, see Section 5.1). Further, suppose that two simulators for ML-Rules are available: `Simulator A` and `Simulator B`. In ML-Rules, the reaction network can be dynamic and might change during runtime. Similarly, the number of species can change during runtime. Therefore, one might conclude that the number of reactions and the number of species are important for the runtime performance of the simulator. In this example, we assume that these features are used to describe a base state. Consequently, the base state space $\Sigma$ is two-dimensional, i.e., $\Sigma = \mathbb{N}^2$, and a base state $\sigma$ is a tuple: $(s, r)$ ($s$ represents the current number of species, $r$ represents the current number of reactions in the reaction network). A base state trajectory therefore is a sequence of these tuples, e.g., $\tau = ((s_1, r_1), (s_2, r_2), \ldots)$.

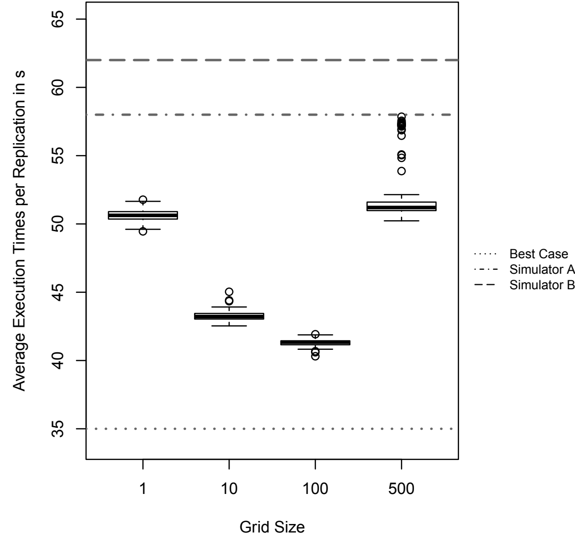



Figure 4.6: Distribution of average runtime per replication of the `Adaptive Simulator` with different state space grid sizes.

Further, the function $p_1 : \Sigma^* \rightarrow S$ shall aggregate a base state trajectory by averaging its tuples, i.e.,

$$p_1(\tau) = (\lfloor \frac{s_1 + s_2 + \dots}{|\tau|} \rfloor, \lfloor \frac{r_1 + r_2 + \dots}{|\tau|} \rfloor).$$

Using these methods, we executed 100 simulation runs (40,000 events per run) with the `Adaptive Simulator` of an ML-Rules benchmark model (see Section 5.7, page 110). Further, the `Adaptive Simulator` is executing these simulation runs sequentially and it is reusing gained knowledge of already simulated replications. To get suitable statistics about the efficiency of the `Adaptive Simulator`, the whole experiment has been repeated 100 times. The `Adaptive Simulator` uses $\epsilon$-decreasing, Q-learning with $\alpha(N[s,a]) = \frac{1}{N[s,a]}, \gamma = 0$, a $log_2 n$ reward of the event throughput and adapts each 1000 simulation events. All experiments of the running examples have been executed with the same machine (Intel Xeon CPU X5690, 48GB RAM, Windows 7 64bit, Java 8). Detailed runtime results of `Simulator A` and `Simulator B` are shown in Figure 4.5. Thus, `Simulator B` is more efficient for the first and third part of the simulation run and `Simulator A` is more efficient for the second and fourth part. For a whole simulation run, both simulators perform similarly on average for a whole simulation run (`Simulator A`: $\approx 58s$, `Simulator B`: $\approx 62s$). Figure 4.7 (top) shows all occurring states based on the aggregation of the observed base states trajectories (left) and the better simulator for each observed state (right). Altogether, more than 400 states can be observed. For each state, Q-learning must learn which simulator

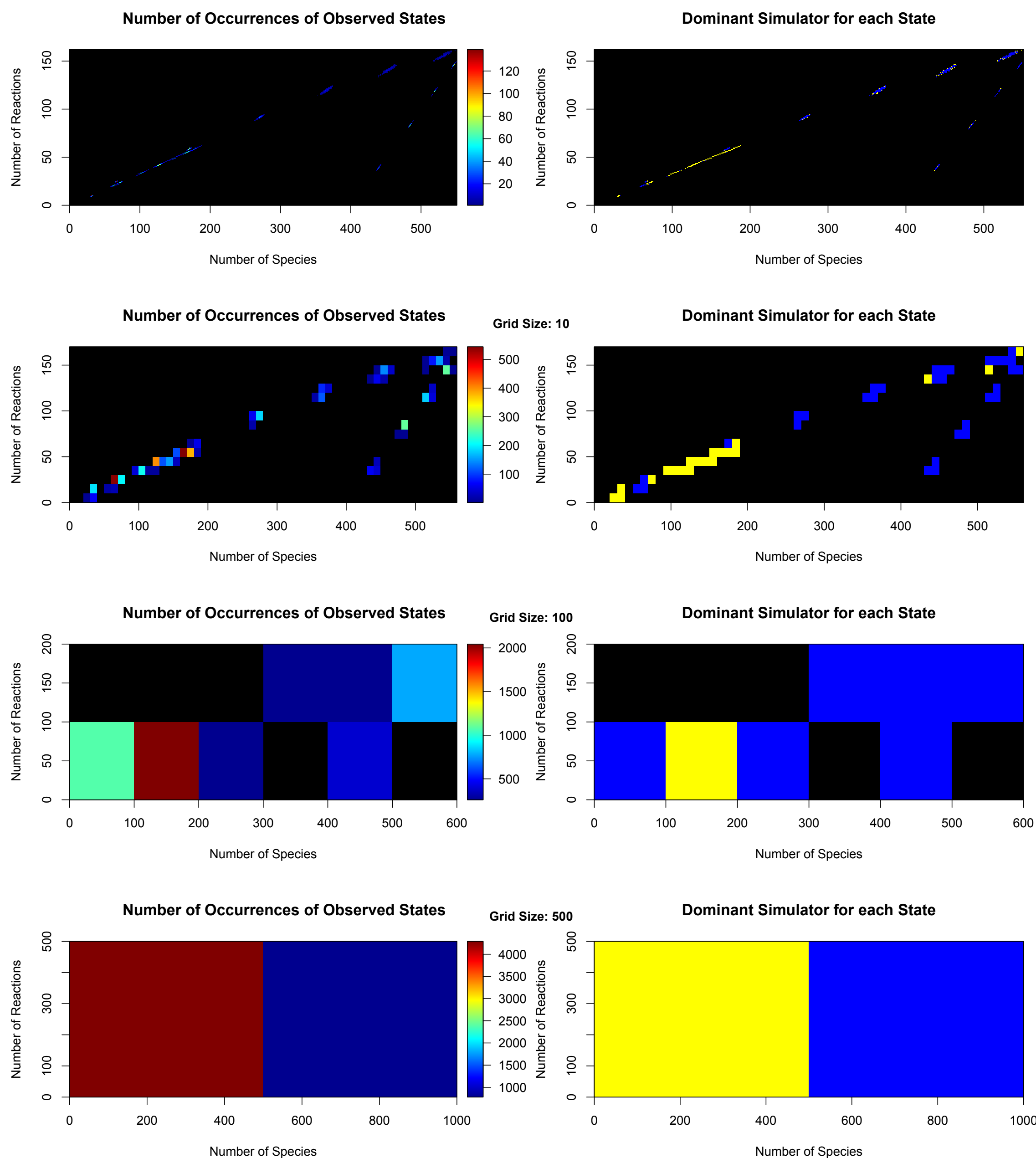



Figure 4.7: Illustration of observed states without generalization (top) and with a grid-based generalization and three different grid sizes. Left: All occurred states for the ML-Rules benchmark model. The color denotes how often a state has been observed. The color black is used for states that have not been observed. Right: For each observed state, the better simulator is shown (yellow = `Simulator A`, blue = `Simulator B`).

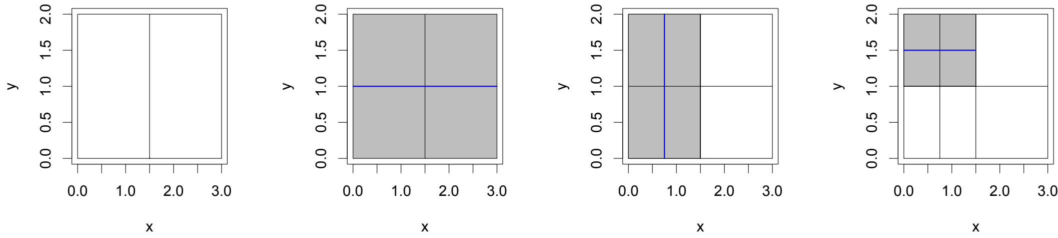



Figure 4.8: Illustration of state splitting (from left to right) by the DBPA. Gray marked states are split at the blue lines, i.e., the midpoint of their longest dimension.

performs better. Therefore, using a generalization method for this example might be effective, since for some areas of the state space, `Simulator A` (yellow) dominates and for other areas `Simulator B` (blue) dominates. Firstly, we applied a regular grid to generalize states with different sizes for each grid cell. Figure 4.7 also shows the observed states and dominant simulators when using a fixed grid for the generalization of states with three different grid sizes (10, 100 and 500). Referring to the results of the `Adaptive Simulator`, the more coarse grained the generalization of the state space, the fewer states are observed and the learning efficiency is increased, see Figure 4.6. In case of a too coarse grained generalization, however, the performance becomes worse as the simulation phases cannot be distinguished suitably anymore. Clearly, using a generalization based on a fixed grid can be beneficial, however, it is not trivial to determine a suitable grid size and different sizes might be suitable for different areas of the state space.

### 4.3.1 Decision Boundary Partitioning

A well-known dynamic partitioning algorithm is the **decision boundary partitioning algorithm (DBPA)** [165]. This algorithm starts with two macro states. At the beginning of each trial or every $n \in \mathbb{N}$ steps the algorithm analyzes all adjacent macro states. The areas of two adjacent macro states $ms_i$ and $ms_j$ are split at the midpoint of their longest dimension if the following three conditions hold. First, based on the current knowledge, the best action in both macro states differ, i.e.,

$$\underset{a \in A}{argmax}\, Q[ms_i, a] \neq \underset{a \in A}{argmax}\, Q[ms_j, a]. \tag{4.2}$$

Second, one absolute difference between the q-values of the best actions $a_i$ and $a_j$ is higher than $\Delta_{min} \in \mathbb{R}$:

$$|Q[ms_i, a_i] - Q[ms_i, a_j]| > \Delta_{min} \vee \; |Q[ms_j, a_i] - Q[ms_j, a_j]| > \Delta_{min}. \tag{4.3}$$

Third, all actions of both states have been visited at least $v_{min}$ times:

$$\forall a \in A : N[ms_i, a] \geq v_{min} \wedge N[ms_j, a] \geq v_{min}. \tag{4.4}$$

The first condition guarantees that areas are only split if there is a change with respect to the best action. The second condition avoids unnecessary splits because probably no benefit can be expected. The third condition is important to avoid splits based on fragile knowledge. Figure 4.8 illustrates how areas of a two dimensional space $[0, 3] \times [0, 2]$ could be split by the DBPA.

However, these conditions do not guarantee a suitable partitioning. For example, if many states occur inside the same area but not inside its neighbors, this algorithm will not split this area although it could be useful. In the worst case, no splits are executed at all because the initial areas have been set poorly.

In principle, the algorithm can be applied to the `Adaptive Simulator` [90]. Each $n \in \mathbb{N}^+$ adaptations, directly after updating the knowledge base (Algorithm 4.1, l.24) a `partitionCheck()` method is called that splits all adjacent regions fulfilling the three splitting conditions. When a region represented by the macro state $m$ is split, two new regions with the macro states $m_1$ and $m_2$ are created that initially use the q-values of $m$, i.e., $\forall a \in A : Q(m_1, a) = Q(m_2, a) = Q(m, a)$. However, the counter matrix $N$ is reset for the new regions, i.e., $\forall a \in A : N(m_1, a) = N(m_2, a) = 0$. Thus, the new regions consider the knowledge of their "parent" region, but they are also willing to explore again.

Regions are split at the midpoint of their longest dimension. Thus, it is necessary to define a range of each dimension. Unfortunately, no minimum and maximum values for each dimension are known by the `Adaptive Simulator` during runtime, i.e., fixed ranges would have to be set manually by the user. Instead, we use the current minimum and maximum values of all occurred states, see Figure 4.9. When a state occurs outside of the current state space area, the area is extended to this state accordingly.

Although the algorithm seems to be promising in general to dynamically partition a state space for reinforcement learning, it turned out that it is difficult to be applied to the `Adaptive Simulator`. Firstly, there are too many parameters that are difficult to configure: the splitting frequency $n$, the minimum reward difference of neighbored states $\delta_{min}$, the minimum number of action selections $v_{min}$, and the initial

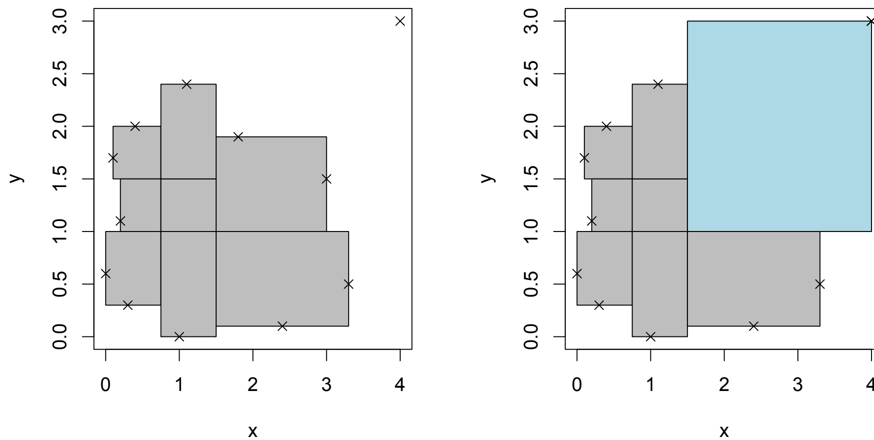



Figure 4.9: Left: A new state (top right) occurs outside the current state space area (grey area) determined by the outermost states for each area. Right: The blue area has been extended to the new state.

partitioning. Especially a suitable initial partitioning is essential for the effectiveness of the algorithm, i.e., if most states occur within the same region and not within its neighbor regions, no splits would be executed at all. In general, the restriction that only two adjacent regions can be split *simultaneously* by fulfilling the requirements hampers the effectiveness of the algorithm. Additional requirements for individual regions to be split should be added. For example, split a region if a specific number of states occurred inside it.

## Running Example Part 2

Continuing the running example, we executed simulation runs with the `Adaptive Simulator` using the DBPA ($n = 5$, $\delta_{min} = 0.01$, $v_{min} = 2$). Figure 4.10 shows three calculated state space representations chosen from the 100 repetitions of the whole simulation experiment: the generalization resulting in the best performance, the generalization resulting in the worst performance and a generalization resulting in an average performance are shown. Figure 4.11 shows the distribution of the average runtime per replication of the repetitions. All in all, in most cases, the `Adaptive Simulator` performs better with the DBPA compared to the fixed grids, i.e., it learns more efficiently. Nevertheless, in some cases, the DBPA fails (see the worst case generalization in Figure 4.10) and only the performance of `Simulator A` is achieved. Generally, we spent much effort to find at least one configuration of

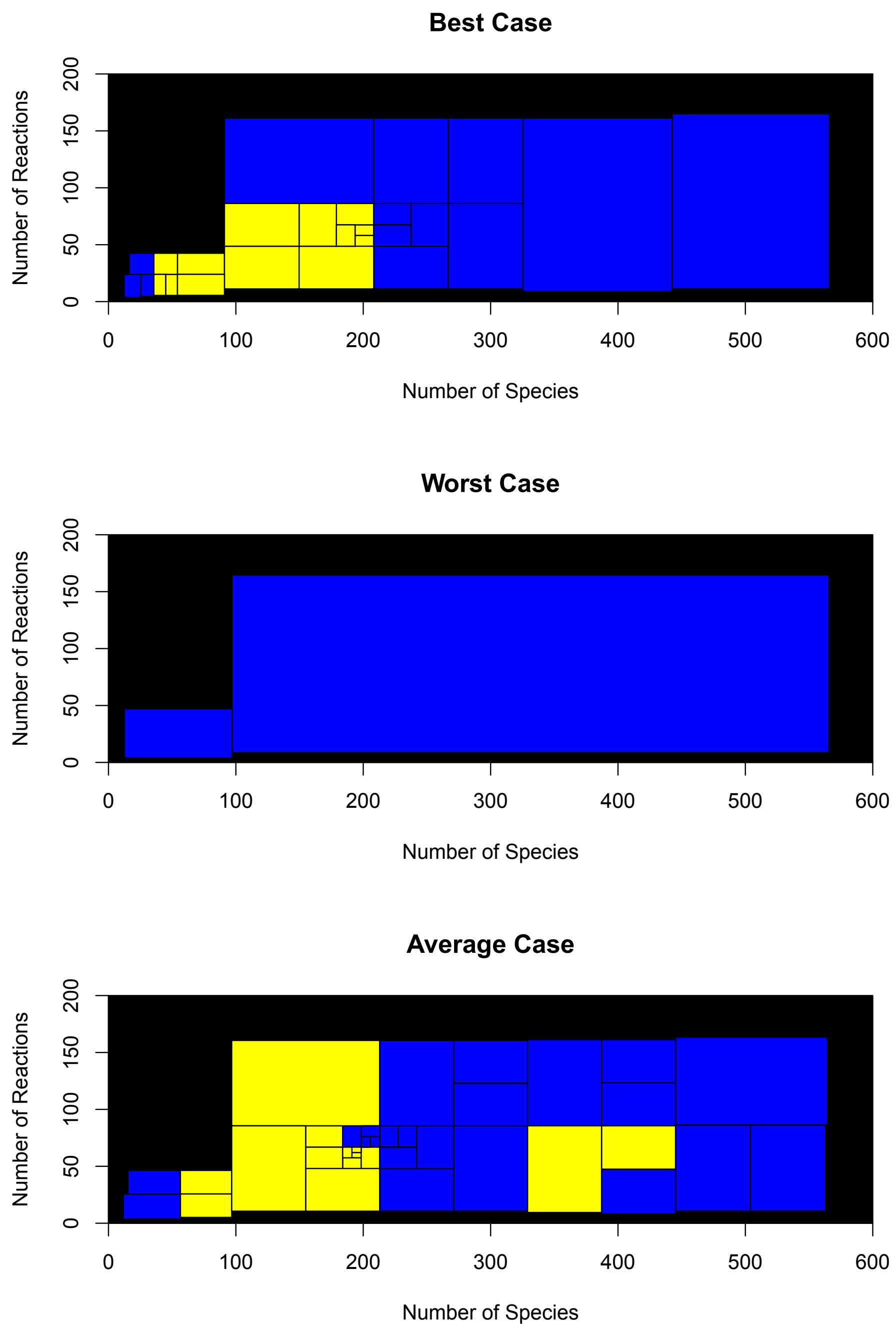



Figure 4.10: Illustration of the created state space representation with the DBPA used by the `Adaptive Simulator` with the ML-Rules benchmark. For each observed state, the better simulator is shown (yellow = `Simulator A`, blue = `Simulator B`).

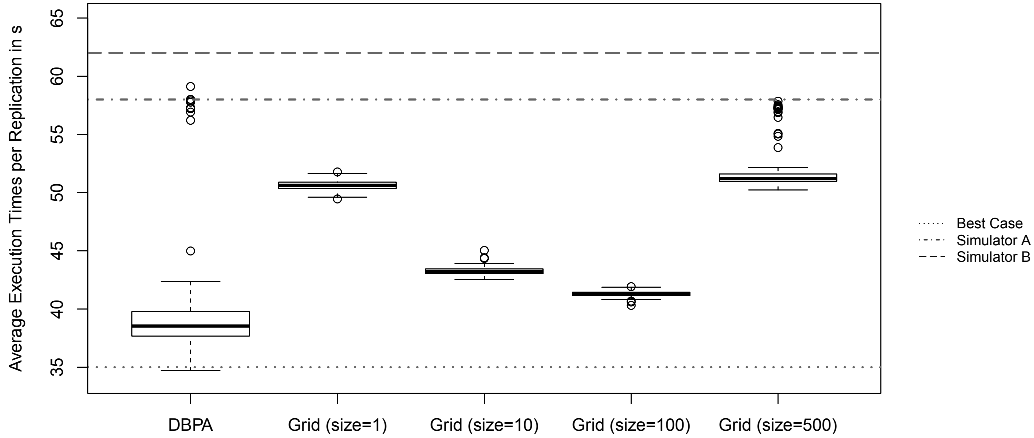



Figure 4.11: Distribution of average runtime per replication of the `Adaptive Simulator` using the DBPA compared to grid-based results.

the DBPA which produced some suitable results — the DBPA has been sensitive to small parameter changes. For most configurations, either too few splits had been executed (important decision boundaries have not been considered) or too many splits had been executed (the learning rate of the algorithm decreased significantly). This configuration challenge contradicts the requirement of the `Adaptive Simulator` to reduce the configuration effort of the user as well as the failure rate of the DBPA motivate the exploration of alternative generalization methods.

## 4.3.2 Adaptive Vector Quantization

Another group of aggregation algorithms uses the idea of the nearest neighbor vector quantization to identify macro states, e.g., the **adaptive vector quantization algorithm (AVQ)** [114]. These algorithms maintain a codebook $CB \subseteq S$ containing specific states that are called codewords. A nearest vector quantizer is used to map a state $s \in S$ onto the nearest codeword $c \in CB$ available in the current codebook, i.e., the nearest neighbor problem must be solved [186]. Basically, this mapping creates a partitioning of the state space into disjoint regions. Figure 4.12 shows how such a partitioning can evolve in form of Voronoi diagrams. In contrast to the DBPA, the areas of the state space created by a codebook can form more complex shapes than hyperrectangles. Algorithm 4.2 outlines the AVQ. To decide whether new codewords shall be added to the codebook, the algorithm uses a concept based on the accumulated reward (*accReward*), that "with respect to a particular action is the sum of the total

**Algorithm 4.2** Outline of the pseudo-code for the AVQ [114].
$Q$: q-value matrix indexed by state $s \in S$ and action $a \in A$.
$N$: matrix of counters for visited $(s, a)$ tuples.
$CB \subseteq S$: codebook. $c, c' \in S$: codeword.
$s, s' \in S$: state. $a, a' \in A$: action. $r \in \mathbb{R}$: reward.

```
1  s := initial(S) // Get initial state
2  c := nearest_codeword(s,CB) // Get codeword for initial state
3  a := next_action(Q,s) // Use MABP to select action
4  accReward := 0
5
6  repeat { // Trial loop
7    execute(a) // Apply action
8    s' := observe() // Observe next state
9    r := reward(s')
10   c' := nearest_codeword(s',CB)
11   a' := next_action(Q,c')
12
13   if (c == c') { // Check if codeword has not changed
14     accReward := accReward + r
15     if (accReward > χ && dist(c',s') > Δ) {
16       CB := CB ∪ {s'}
17       c' := s'
18       update_A(Q,N,c',a,r) // Update knowledge base
19       accReward := 0
20     } else {
21       a' := a // Reuse previous action
22     }
23   } else {
24     Q(c,a) := Q(c,a) + α·[r + γ·max_a Q(c',a) − Q(c,a)] // Q-learning
25     accReward := 0
26   }
27   c := c', a := a'
28 } until end of trial
29
30 //Merging process
31 for (c ∈ CB) {
32   c' := nearest_neighbor(c,CB)
33   if ((∑_{a∈A}(Q[c,a] − Q[c',a])²) ÷ |A| < ρ) {
34     CB := (CB\{c,c'}) ∪ {(c+c')/2}
35     update_B(Q,N,c,c') // Update knowledge base
36   }
37 }
```

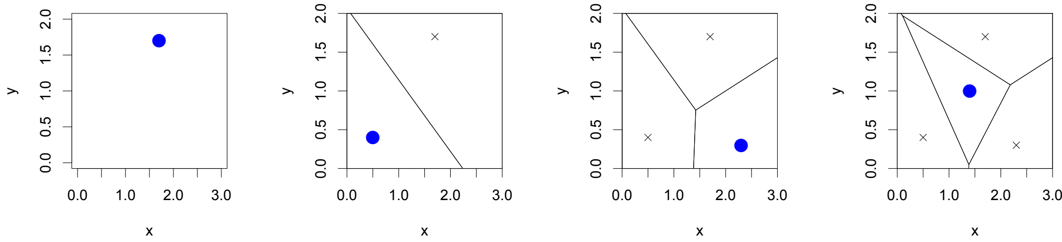



Figure 4.12: Illustration of codebook extension by the AVQ. Codewords (blue circles) are added from left to right and the state space partitioning is adapted accordingly.

rewards received by continuously taking the same action within a particular cell" [114]. The accumulated reward increases if the observed states map to the same codeword (l.13-14). Further, if the accumulated reward exceeds a threshold $\chi \in \mathbb{R}$, the current state is added to the codebook and the matrix of q-values and selection counters is updated (l.18). Otherwise, the previously executed action is selected again (l.21). Thus, as long as the observed states map to the same codeword and the accumulated reward is not large enough, the same action is used repetitively. The q-values are not updated in this case; they are only updated if two successive observed states map to different codewords (l.24).

After finishing a trial, in contrast to the DBPA, a merging process is executed (l.31-37), see Figure 4.13. Here, for every nearest neighbor pair of codewords $(c, c')$, it is checked whether the mean squared difference of their q-values is smaller than a threshold $\rho \in \mathbb{R}$. In this case, the codewords $c$ and $c'$ are removed from the codebook, a new codeword $[(c + c')/2] \in S$ is added to the codebook (l.34), and the knowledge base is updated properly (l.35).

We integrated the AVQ to the `Adaptive Simulator` [79]. The codebook $CB$ of the algorithm can directly be used as the set of macro states, i.e., $M = CB$. However, the concept of the accumulated reward is not applied directly to the `Adaptive Simulator`. First, the codebook is only extended if successive states often map to the same codewords (otherwise, it is unlikely that $accReward > \chi$). This assumption might be useful in various scenarios, e.g., in the maze scenario in which the agent cannot "beam" itself trough the state space. However, successive observed states of the `Adaptive Simulator` can be completely different, so that it is possible that they will not frequently map to the same codeword. In this case, the codebook would quickly stop to grow as the accumulated reward is always set to 0 after a codeword change. Second, using the same action repetitively if no codeword change has been observed can directly reduce the efficiency of the adaptive simulator since it performs

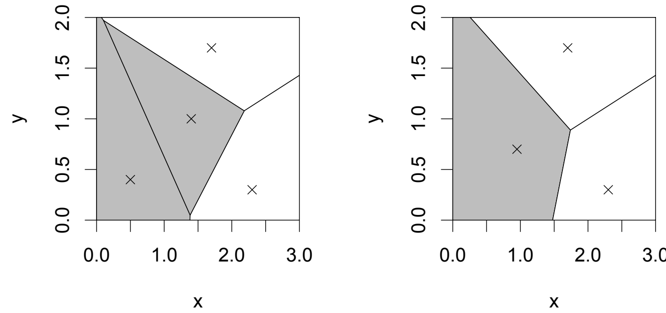



Figure 4.13: Illustration how two codewords are merged by the AVQ. Left: The codewords of the gray marked areas shall be merged. Right: A new codeword lying between both old codewords has been added to the codebook.

the learning *online* and has no separated learning phase. Third, the knowledge base is only updated when a new codeword is added, so that all rewards received during a phase of *accReward* increases are ignored for the q-values. Especially when the performance varies a lot, this can reduce the learning effectiveness.

Consequently, based on these arguments, we replaced the concept of the accumulated reward with a condition inspired by the DBPA: a state $s$ mapped to the codeword $c$ is added to the codebook if the absolute difference of the current reward and the last reward achieved by the same action for any other state $s'$ mapped to $c$ is higher than a threshold $\alpha \in \mathbb{R}$. Generally speaking, a state is added to the codebook if the rewards of an action differ significantly within its region. In our first approach of the algorithm [136], see Algorithm 4.3, we also use a minimum distance condition, i.e., $dist(s, c) > \Delta$, and the merging routine is executed each $md \in \mathbb{N}$ adaptations. Altogether, the algorithm uses five parameters:

- $\alpha \in \mathbb{R}^+$: Minimum difference of two rewards for the same action and region to add a state of this region to the codebook.
- $\Delta \in \mathbb{R}^+$: Minimum distance of two codewords.
- $v_{min} \in \mathbb{N}^+$: Another trigger to add a state to the codebook, i.e., if a region has been visited $v_{min}$ times, the current observed state of this region is added to the codebook.
- $md \in \mathbb{N}^+$: Frequency of merging routine as defined in Algorithm 4.5.
- $\rho \in \mathbb{R}^+$: Maximum mean squared reward difference of nearest neighbor codewords to be merged, see Algorithm 4.5, l. 3.

**Algorithm 4.3** Pseudo-code of our initial version of the AVQ within the `Adaptive Simulator`.
$Q$: q-value matrix indexed by aggregated state $s \in S$ and action $a \in A$.
$N$: matrix of counters for visited $(s, a)$ tuples.
$s \in S$: current aggregated state. $m \in M$: macro state of $s$.
$a \in A$: current action. $r \in \mathbb{R}$: current reward.
$L : M \times A \rightarrow \mathbb{R}$: matrix containing the last rewards indexed by macro states and actions.
$v_{min}$: minimum number of selections of an action $a$ for a state $s$ to add $s$ to the codebook.
*counter*: global counter of `partitionCheck()` calls.

```
1  counter := counter + 1
2  m := nearestNeighbor(M, s)
3
4  // check to add s to codebook
5  if ((|L[m, a] − r| > α || Σ_{a_i∈A} N[s, a_i] > v_min) && dist(m, s) > Δ) {
6     M := M ∪ {s}
7     for (a_i ∈ A) {
8        Q[s, a_i] := Q[m, a_i]
9        N[s, a_i] := 0
10    }
11    m := s
12 }
13 L[m, a] := r
14
15 // start merge routine
16 if (counter > md) {
17    merge() // see Algorithm 4.5
18    counter = 0
19 }
20
21 return m
```

To avoid an additional configuration effort, we used ParamILS [94] to find one configuration of this AVQ variant that works well for a set of benchmark scenarios [136]. ParamILS is an automatic configuration search framework available in JAMES II [44] used to systematically search a configuration space. Basically, ParamILS starts with a manually chosen configuration $c_0$ and $r \in \mathbb{N}$ randomly chosen configurations $(c_1, \ldots, c_r)$ and determines the best performing configuration $c_i$ of these configurations. Next, a local search is started from $c_i$, i.e., the best performing configuration $c_j$ of $c_i$ and

**Algorithm 4.4** Pseudo-code for $p_2$ based on our variant of the AVQ algorithm, see [79]. The merging routine is executed separately at the end of each simulation run (see Algorithm 4.5).
$Q$: q-value matrix indexed by aggregated state $s \in S$ and action $a \in A$.
$N$: matrix of counters for visited $(s, a)$ tuples.
$s \in S$: current aggregated state. $m \in M$: macro state of $s$.
$a \in A$: current action. $r \in \mathbb{R}$: current reward.
$L : M \times A \to \mathbb{R}$: matrix containing the last rewards indexed by macro states and actions.

```
1  m := nearestNeighbor(M, s)
2  if (|M| < c_max && |L[m, a] − r| > α) {
3    M := M ∪ {s}
4    for (a_i ∈ A) {
5      Q[s, a_i] := Q[m, a_i]
6      N[s, a_i] := 0
7    }
8    m := s
9  }
10 L[m, a] := r
11 return m
```

its neighbor configurations is determined and the process is repeated with $c_i = c_j$ until no improvement can be observed anymore. Additionally, during the local search ParamILS restarts with a random configuration with probability $p_{restart}$. Referring to the AVQ and the benchmark scenarios, we have not been able to determine one configuration with satisfying results for all scenarios.

Based on these results, we changed the algorithm [79], see Algorithm 4.4 and Algorithm 4.5. The minimum distance $\delta$ has been removed. This parameter is used to avoid arbitrary small regions. It is challenging to configure this parameter for a concrete problem because the scales of the state space dimensions are typically not known initially. Also, a tiny distance might be useful for some regions, whereas for other regions it might be unnecessary. We replaced the minimum distance restriction with an approach limiting the size of the codebook with a parameter $c_{max} \in \mathbb{N}^+$. Basically, the smaller $c_{max}$ is chosen, the fewer simulation phases can be distinguished, but also the exploration effort is reduced. This effect can be advantageous if there is not much time available for exploration, e.g., in case a user is developing a model and often starts only one simulation run with the current model, then changes it, then starts one simulation run again, etc. Further, we removed the parameter $v_{min}$. Originally, the purpose of $v_{min}$ was to avoid a situation in which a region is not split although it

**Algorithm 4.5** Pseudo-code for our AVQ merging method, see [79].
$Q$: q-value matrix indexed by aggregated state $s \in S$ and action $a \in A$.
$N$: matrix of counters for visited $(s, a)$ tuples.
$m_1, m_2 \in M$: macro states to be merged. $m_{new} \in M$: new macro state.

```
1  for (m_1 ∈ M) { // newly merged macro states are not considered
2    m_2 := nearestNeighbor(M, m_1)
3    if ((∑_{a_i∈A}(Q[m_1, a_i] − Q[m_2, a_i])^2) ÷ |A| < ρ) {
4      m_new := (m_1 + m_2)/2
5      M := (M\{m_1, m_2}) ∪ {m_new]}
6      for (a_i ∈ A) {
7        Q[m_new, a_i] := (Q[m_1,a_i]·N[m_1,a_i]+Q[m_2,a_i]·N[m_2,a_i]) / (N[m_1,a_i]+N[m_2,a_i])
8        N[m_new, a_i] := N[m_1, a_i] + N[m_2, a_i]
9      }
10   }
11 }
```

is probably interesting because many states occur in this region. It was motivated by the worst case scenarios observed with the DBPA, where sometimes no splits are executed at all since all observed states lie in one region, see Section 4.3.1. However, the motivation of this parameter is misleading for our AVQ variant as a region that is visited frequently is likely to be refined if there are reward variances. Further, in case that there are no reward differences in a region, no unnecessary refinement takes place. Moreover, the merging frequency $md$ has been removed and replaced by the heuristic to execute the merging process after each replication execution. Further, to couple the threshold to add a codeword to the codebook with the threshold to merge two codewords, we set $\rho = \alpha \times \alpha$. Altogether, the revised variant of the algorithm only has left two parameters: $\alpha$ and $c_{max}$.

## Running Example Part 3

Figure 4.14 illustrates three created state space generalizations for the benchmark ML-Rules model with the `Adaptive Simulator` and the AVQ algorithm ($c_{max} = 100$, $\alpha = ln_2 1.5$). The best performing generalization, the worst created generalization, and a generalization resulting in average results are shown. By $\alpha = ln_2 1.5$ and a $log_2$ reward function of the event throughput, a throughput difference of at least 50% must occur so that a codeword is added to the codebook. Further, the average replication runtimes are shown in Figure 4.15. Compared to the DBPA, the AVQ performs worse for most cases, but it does never fail like the DBPA. Further, determining the

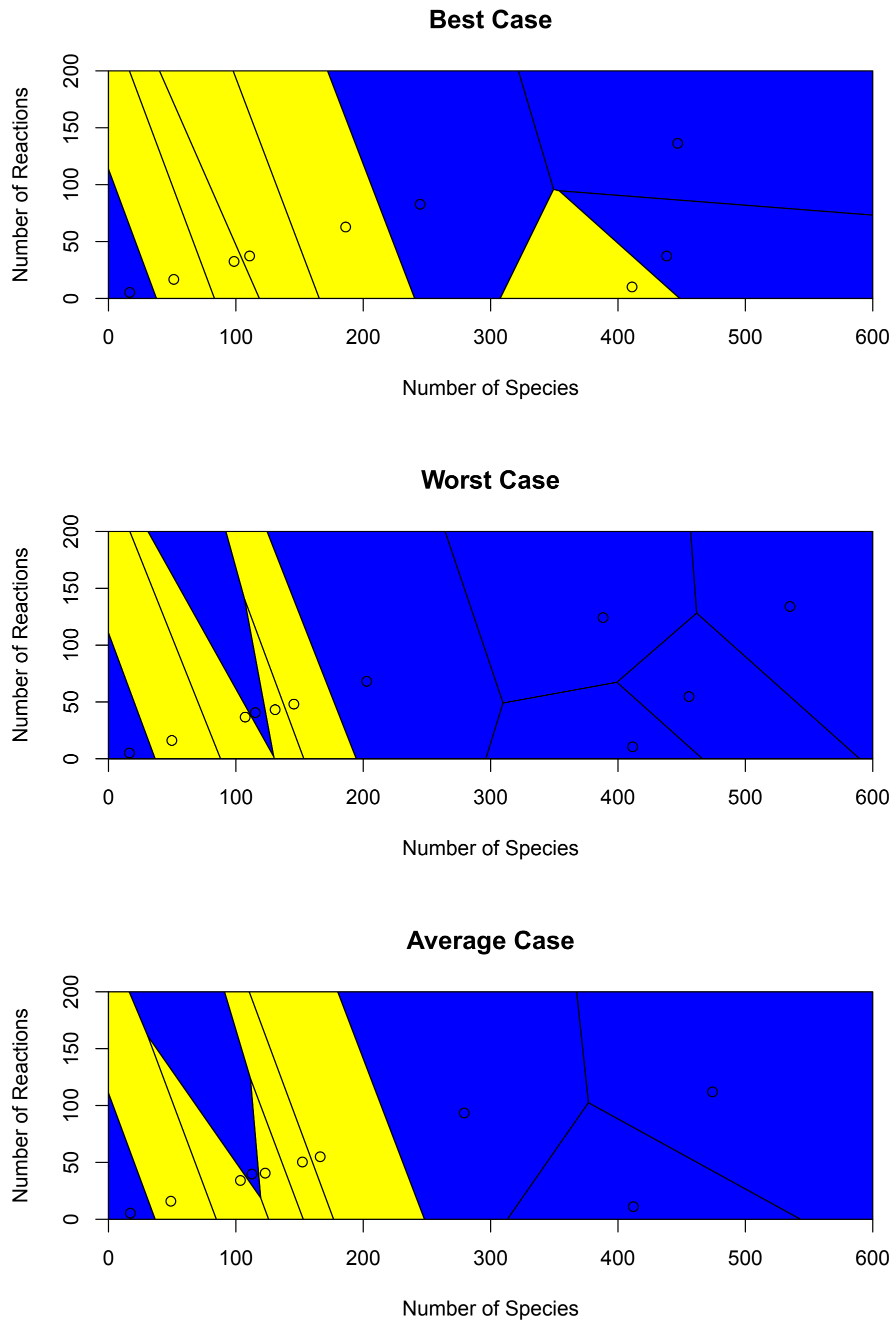



Figure 4.14: Illustration of the created state space representation with the AVQ used by the `Adaptive Simulator` with the ML-Rules benchmark. For each observed state, the better simulator is shown (yellow = `Simulator A`, blue = `Simulator B`).

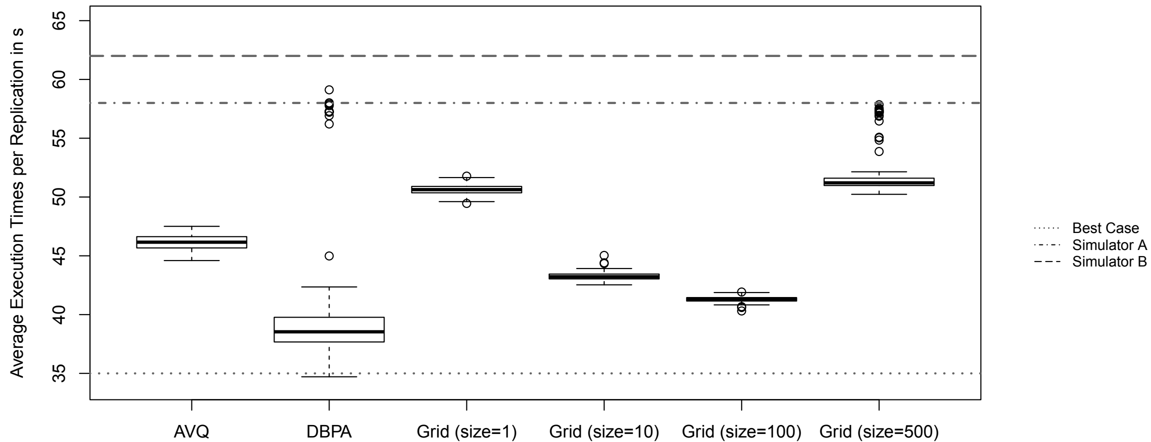



Figure 4.15: Distribution of average runtime per replication of the `Adaptive Simulator` using the AVQ compared to the results usng the DBPA and the grid-based generalization (grid size = 10).

used configuration of the AVQ has been easier compared to the determination of the used DBPA configuration and the AVQ also has achieved good results with various configurations. Therefore, we conclude it to be more robust compared to the DBPA.

## 4.4 Adaptation Conditions

The challenge to define suitable adaptation conditions and frequencies is well-known in the domain of adaptive software and refers to its basic facets, e.g., see the **changes** facet defined by Andersson [3], see Section 3. Basically, developers of adaptive software should always analyze causes and conditions for adaptations and they should consider whether causes for adaptations can be foreseen. The overall aim of suitable adaptation conditions is that adaptations are only executed if they are somehow beneficial, e.g., if they exchange faulty components, improve the performance, or enable the software to gain useful knowledge like performance data.

Referring to the `Adaptive Simulator`, various opportunities exist to define adaptation conditions. For example, specific model events (e.g., the start of a fire or the start of an epidemic) or specific changes in model dynamics could be used to trigger adaptations. On the one hand, this approach enables a tailored definition of adaptation conditions that might be useful if, e.g., rare events dramatically change the model dynamics and this is known beforehand. On the other hand, this approach

is strongly application-dependent: the user must identify events and properties of the model that might be suitable to be used as adaptation conditions.

A more generic but still simple idea is to execute adaptations regularly with fixed intervals based either on the wall-clock time, simulation time, or the number of events that has been processed. These variants are also difficult to be applied to the `Adaptive Simulator` as they require the user to configure a suitable interval length that will likely be application-dependent. Further, it might be beneficial to adapt frequently during some parts of the simulation run and rarely during other parts.

In any case, adaptations should not be triggered too often. Besides the overhead of executing the adaptation process itself, this may also bias learning: simulators can have warm-up phases, so that advantages of these algorithms are only noticeable after having processed many simulation events in a row. If adaptations are executed too often, the actual performance of such algorithms would never be noticed, although they might improve the overall performance significantly.

In our first approach to solve the adaptation condition problem, we considered these thoughts by integrating adaptation conditions into the actions, i.e., an action is represented by a tuple consisting of a simulator configuration and an adaptation condition [75]. Thus, choosing an action determines which configuration shall be used until the next adaptation, and also under which conditions the next adaptation is triggered. Consequently, with a set of suitable adaptation conditions, the `Adaptive Simulator` automatically learns a good trade-off between minimizing the number of adaptations and using the best simulator for each simulation phase. This approach of learning two things is similar to the notion of subroutines that are used in hierarchical reinforcement learning [10]. Nevertheless, by using this approach, the number of available actions is determined by the cross product of all available simulator configurations and all adaptation conditions, so that each adaptation condition reduces the learning efficiency as more actions have to be explored [75]. Further, either many adaptation conditions have to be considered to ensure that suitable conditions are always available, or application-dependent knowledge is applied to restrict the number of adaptation conditions to a feasible small set. The first approach would increase the action set unacceptably. The second approach requires application-dependent knowledge. Thus, both options are not suitable to be applied to the `Adaptive Simulator`. Besides the described possibilities to trigger adaptations for the `Adaptive Simulator`, we also followed a more generic and sophisticated approach by using Bayesian online changepoint detection that is described in the next section.

### 4.4.1 Changepoint Detection for Adaptive Simulation Algorithms

Bayesian online changepoint detection as defined by Adams and MacKay identifies abrupt variations in a data sequence by only considering previously observed data points [1]. Online approaches do not segment data retrospectively, but make predictions to decide whether a changepoint occurred or not. The algorithm assumes that a sequence of observations can be divided into disjunct partitions $p_1, p_2, \ldots$ that generate data points $x_1, x_2, \ldots$ from the same probability distribution $P(\eta_{p_i})$ with different parameters. The idea of the algorithm is to estimate the probability distribution of the time since the last changepoint: the "run length" of the current partition, see Figure 4.16. For each time point $t$, every possible run length is associated with a probability $p(run_t|x_{1:t})$, where $x_{1:t}$ denotes the data points $x_1, \ldots, x_t$. The probability distribution $p(run_t|x_{1:t})$ is computed by

$$p(run_t|x_{1:t}) = \frac{p(run_t, x_{1:t})}{p(x_{1:t})}, \tag{4.5}$$

where

$$p(x_{1:t}) = \sum_{run_t} p(run_t, x_{1:t}), \tag{4.6}$$

and

$$p(run_t, x_{1:t}) = \sum_{run_{t-1}} p(run_t|run_{t-1}) \cdot p(x_t|x_{t-1}^{(run_{t-1})}) \cdot p(run_{t-1}, x_{1:t-1}). \tag{4.7}$$

For all $run_t = n > 0$, the sum of Equation 4.7 reduces to exactly one summand with $run_{t-1} = n - 1$, because no other previous run lengths are possible with $run_t = n > 0$, see the bottom plot of Figure 4.16. Only if $run_t = 0$, several summands have to be considered. The probability $p(run_t|run_{t-1})$ is the general probability for a changepoint that is independent from the observed data points and that can be described by a *hazard rate*:

$$p(run_t|run_{t-1}) = \begin{cases} 1-h, & \text{if } run_t = run_{t-1}+1 \\ h, & \text{if } run_t = 0. \end{cases} \tag{4.8}$$

In the most simple case, the hazard rate is a constant probability. The term $p(x_t|x_{t-1}^{(run_{t-1})})$ denotes the predictive distribution of the current data point $x_t$ based on the last $run_{t-1}$ data points: $x_{t-1}^{(run_{t-1})} = x_{t-run_{t-1}:t}$. This distribution can be recursively

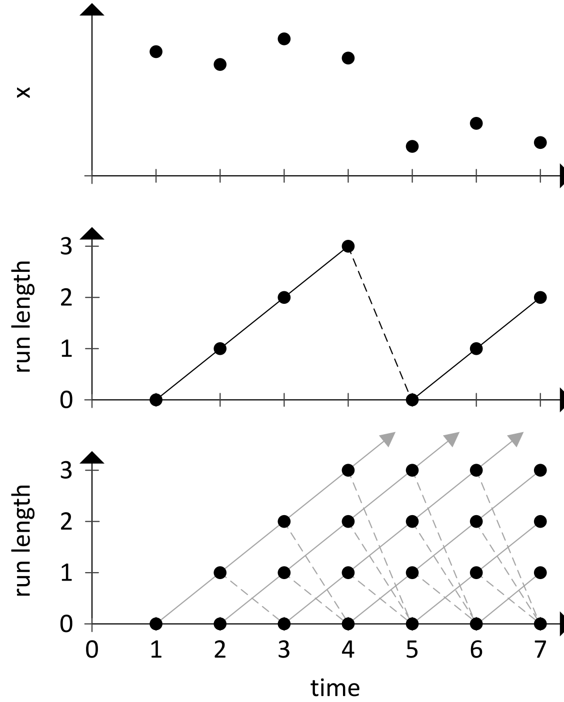



Figure 4.16: Illustration of the run length concept of the Bayesian online changepoint detection algorithm as described by Adams and MayKay [1]. Top: Generated data points with one changepoint between the fourth and fifth value. Middle: The run length of the current partition for each time point. Bottom: Possible run lengths $run_t$ at each time point $t$.

calculated by marginalization:

$$p(x_t|x_{1:t-1}) = \sum_{run_{t-1}} p(x_t|x_{t-1}^{run_{t-1}}) \cdot p(run_{t-1}|x_{1:t-1}). \tag{4.9}$$

The changepoint detection approach can be used by the `Adaptive Simulator` to determine useful adaptation points [164, 80]. The basic idea is to calculate the performance of the current internal simulator regularly, i.e., the reward $r \in \mathbb{R}$ is not only calculated when an adaptation shall be executed, but also regularly during the simulation loop of Algorithm 4.1. These reward values are interpreted as data points for the changepoint detection algorithm and the `Adaptive Simulator` executes an adaptation after it identifies a changepoint within the reward values. The assumption of this approach is that the performance of simulators changes when different phases of a simulation run are executed.

Algorithm 4.6 shows the concept of our approach. Before applying the changepoint detection algorithm, two conditions are checked. First, the length $|\tau|$ of the current

base state trajectory $\tau$ is checked (l. 2). For this, we introduce a parameter $a_{min} \in \mathbb{N}^+$. By using this parameter, it is guaranteed that an adaptation can only be executed at most each $a_{min}$ simulation events. Assigning $a_{min} > 1$ becomes important if the execution time of a simulation event is small, e.g., a few nano seconds, because firstly the noise of the computation time would make it impracticable to infer useful adaptation points and secondly the effort to compute the changepoints would be more costly than calculating the simulation events.

Next, the exploration rate of the used policy is used to enforce adaptations (l. 5-6). Therefore, the used policy must provide the possibility to calculate the exploration rate. This is trivial for simple policies like $\epsilon$-greedy or $\epsilon$-decreasing, however, for other policies like `Interval Estimation` [101], this might not be possible. Using the exploration rate is done to balance the speed-up induced by adaptation and the opportunities of learning against the effort required by the learning algorithm. Generally speaking, if $f$ explores with a specific probability, it is suitable to enforce an adaptation with the same probability to update the utility of a simulator, because this simulator is probably chosen only to explore its utility and not due to its actual utility. Without such enforced adaptations, it can happen that too few adaptations are executed to gain sufficient performance data. For example, in case that a simulation run has always the same computational demands, so that the performance is fairly constant, no adaptations would be executed at all. Therefore, it would be possible that a bad-performing simulator is used to execute the complete simulation run.[3]

If both conditions do not result in a method exit, the current reward $r_t$ for the last $a_{min}$ base states is calculated (l. 10). Afterward, the probability $P(run_t = 0, r_{1:t})$ is computed by summing the probabilities of a changepoint for all run lengths $0, \ldots, t-1$ (l. 13). For each run length $run \in \{0, \ldots, t-1\}$, this probability is the product of the probability to observe this run length ($P(run, r_{1:t-1})$), the hazard rate $h \in [0, 1]$, and the probability to observe the current reward $r_t$ based on the generative probability distribution $P_g$ approximated by the last $run$ rewards ($P_g(r_t|r_{t-run:(t-1)})$).

The hazard rate represents the probability of a changepoint that is typically independent from the observed data points. Either, it is assumed that the hazard rate is fixed for all phases of the generative process or it is assumed that different phases of the process with individual constant hazard rates exist. A fixed hazard rate is either set initially once or learned incrementally e.g., by counting the number of steps and changepoints. Sophisticated hazard rate algorithms exist to deal with changing hazard rates [200]. The `Adaptive Simulator` probably has to deal with changing hazard rates, i.e., the probability of a phase change during a simulation run will not be constant. Nevertheless, applying such algorithms itself is a complex task as they

[3]This behavior corresponds to the behavior of the `AdaptiveSimulationRunner`.

**Algorithm 4.6** Pseudo-code for the `adaptationCondition(`$\tau$`)` method (see Algorithm 4.1, l. 11). The changepoint detection bases on the algorithm described in [1].
$\tau \in \Sigma^*$: current base state trajectory (seq. of base states).
$f : S \times \mathbb{R}^{|S|\times|A|} \times \mathbb{N}^{|S|\times|A|} \to A$ (action selection policy).
$R$: reward function.
$a_{min}$: minimum adaptation interval.
$P_g$: generative probability distribution.
$r_t$: current reward. $r_{1:t-1}$: previous rewards.
$h$: constant hazard probability.
$P_{min}$:minimum probability threshold of no changepoint

```
1  // Check minimal interval length
2  if (|τ| = 0 || |τ| mod a_min ≠ 0 )
3    return false
4  // Check for enforced adaptation
5  if (exploration_rate(f) > x ∈ U(0,1))
6    return true
7  // Compute r_t based on last a_min base states
8  r_t := R(σ_{|τ|-a_min+1} ... σ_{|τ|})
9  // Compute probability of run_t = 0
10 P(run_t = 0, r_{1:t}) := Σ_{run_{t-1} ∈ {0...t-1}} h · P_g(r_t | r_{t-run_{t-1}:(t-1)}) · P(run_{t-1}, r_{1:t-1})
11 // Compute probabilities of run_t ∈ {1...t}
12 for (run_t ∈ {1...t})
13   P(run_t, r_{1:t}) := (1 - h) · P_g(r_t | r_{t-(run_t-1):(t-1)}) · P(run_t - 1, r_{1:(t-1)})
14 // Marginalization
15 P(r_{1:t}) := Σ_{run_{t-1} ∈ {0...t}} P(run_{t-1}, r_{1:t})
16 // Probability of no changepoint so far
17 P(run_t = t | r_{1:t}) := P(run_t, r_{1:t}) / P(r_{1:t})
18 // Update generative distribution (see Equation 4.10)
19 P_g := update(P_g, r_t)
20
21 if (P(run_t = t, r_{1:t}) < P_min)
22   return true
23 else
24   return false
```

add another layer of adaptivity which must be learned, configured, computed, and evaluated. Thus, so far, we only consider constant hazard rates.

For the generative probability distribution $P_g$, we choose a normal distribution, because we assume that the performance of a simulation algorithm is normal distributed due to several independent influences within a phase of the simulation with similar

computational demands. As described by Adams and MacKay, the changepoint detection algorithm assumes that only the parameters of the generative probability distribution $P_g$ change at a changepoint, but not the type of the distribution itself. As we assume the generative probability distribution $P_g$ to be a normal distribution, it is reasonable to use the Normal-Inverse-Gamma distribution $NIG(\mu, \nu, \alpha, \beta)$ that is the conjugate prior of the normal distribution to estimate the unknown mean and unknown variance [9, p. 185-189]. For a Normal-Inverse-Gamma prior $NIG(\mu, \nu, \alpha, \beta)$, the posterior after observing $r_t$ is also a Normal-Inverse-Gamma $NIG(\mu', \nu', \alpha', \beta')$ distribution with

$$\mu' = \frac{\nu \cdot \mu + r_t}{\nu + 1}, \qquad \nu' = \nu + 1, \qquad \alpha' = \alpha + \frac{1}{2}, \qquad \beta' = \beta + \frac{\nu \cdot (r_t - \mu)^2}{2 \cdot (\nu + 1)}. \tag{4.10}$$

Thus, only these simple calculations have to be computed to update the distribution after observing a new data point — represented by the method `update()` (l. 19). Initially, we use $\mu = r_0$, $\nu = 1$, $\alpha = 1$, and $\beta = 1$. While observing more and more data, these distributions become more accurate and the normal distribution of the generative probability distribution is approximated better and better. For a normal distribution with unknown mean and variance, the posterior predictive distribution to calculate the probability of a new data point $p(r_t | r_{1:t-1})$ is a Student's t-distribution

$$t_{2\alpha}\left(r_t | \mu, \frac{\beta \cdot (\nu + 1)}{\alpha \cdot \nu}\right) \tag{4.11}$$

that is calculated by

$$t_n(r_t | \mu, \sigma^2) = \frac{\Gamma(\frac{n+1}{2})}{\sqrt{n \cdot \pi \cdot \sigma^2} \cdot \Gamma(\frac{n}{2})} \cdot \left(1 + \frac{1}{n} \cdot \frac{(r_t - \mu)^2}{\sigma^2}\right)^{-(\frac{n+1}{2})}, \tag{4.12}$$

where $\Gamma(r_t) = \int_0^\infty t^{r_t - 1} e^{-t} dt$ is the gamma function ($\Gamma(r_t) = (r_t - 1)!$ if $r_t$ is a positive integer). Since $\Gamma(r_t)$ is complex to compute for large $n$, the Student-t-distribution should be approximated by the Normal-distribution with more than 30 observations.

Next, the probabilities $P(run_t, r_{1:t})$ for each $run_t \in \{1 \dots t\}$ are computed (l. 12-13), i.e., the probabilities that no changepoint happened. Since it is only possible to reach a run length $run_t \in \{1 \dots t\}$ from the run length $run_t - 1$, no sum is needed here to compute the probability $P(run_t, r_{1:t})$. Using the computed probabilities for all run lengths $run_t \in \{0 \dots t\}$, the probability $P(r_{1:t})$ to observe the given data points can be computed by marginalization (l. 15). After calculating the probabilities for all possible run lengths and the probability of the given data points, this data can be used to compute the probability of no changepoint since the first data point, i.e.,

$P(run_t = t|r_{1:t})$ (l. 17). This probability can be used by the `Adaptive Simulator` to decide whether to execute an adaptation or not (l. 21-24), i.e., if it is smaller than $P_{min} \in [0, 1]$, an adaptation is executed. This approach is suitable for the `Adaptive Simulator` as it is not interested in the concrete moment of the last changepoint, but it is only interested whether a changepoint happened at all in the past.

If an adaptation is executed, it can be assumed that a changepoint occurred by the changes of the simulator, so that $P(run_t = 0|r_{1:t}) = 1$. Consequently, the changepoint detection algorithm can be reinitialized after an adaptation, which significantly simplifies its computational complexity as no data points before the adaptation have to be considered any longer. A further idea to improve the performance of the algorithm suggested by Adams and MayKay is to ignore all run lengths $run_t$ for which the probability $P(run_t, r_{1:t})$ becomes small, e.g., smaller than $10^{-5}$ [1]. In [80], we propose a similar idea that restricts the number of considered run lengths to $\delta \in \mathbb{N}^+$, i.e., when a new run length shall be considered, the run length with the lowest probability is removed from the list of considered run lengths. Analog to sophisticated hazard rates, sophisticated algorithms exist also here to deal with this problem, e.g., the pruning algorithm developed by Wilson et al. merges similar run lengths with similar probability distributions [200].

## 4.5 Implementation & Integration in JAMES II

The `Adaptive Simulator` is integrated into the modeling and simulation framework JAMES II [87] and therefore implemented in Java. Following the plugin concept of JAMES II (see Section 2.5.1), we developed a plugin-based architecture for the `Adaptive Simulator` that delegates the most important tasks to plugins, see Figure 4.17. The class `AdaptiveSimulator` itself is realized as a plugin of the plugin type `Processor` that defines the interface for all simulators in JAMES II. As shown in Figure 4.1, the `Adaptive Simulator` follows the wrapper pattern — it uses an `internalSimulator` to compute the actual simulation and adapts or exchanges the `internalSimulator` as needed. Consequently, with respect to JAMES II, it fulfills the same contract as all other simulators (e.g., regarding stopping criteria or observation components), and thus can be used transparently.

A base state $\tau \in \Sigma^*$ is simply represented by a list of `State` objects that contain a map of key-value pairs. Values of these pairs are of type `Object`, so that in principle arbitrary data can be stored in these pairs. An action $a \in A$ is defined by a `ParameterBlock` and the action set is simply represented by a list of `ParameterBlock` objects. This list is created automatically by the `AdaptiveSimulatorFactory` while creating an `AdaptiveSimulator` object. Here, the class `SelectionTreeSet` of the

SASF framework is reused, which contains methods to traverse all available plugins and to create all possible plugin hierarchy combinations for the simulators. In the SASF framework, for each primitive parameter of the plugins, the default value is used to create all the plugin combinations. We extended the XML scheme for the description of plugins in JAMES II to also support a set of configurations for the primitive parameters. Before, it was only possible to define *one* default value for each plugin parameter that is used to create the action set as follows:

```
<parameter name="flag" type="java.lang.Boolean"
      default="false">
  <description>very important flag</description>
</parameter>
```

We added a new tag `configuration` that allows to specify configurations for all primitive parameter values:

```
<configuration>
  <parameterValue name="flag" value="false">
  </parameterValue>
</configuration>
<configuration>
  <parameterValue name="flag" value="true">
  </parameterValue>
</configuration>
```

This approach allows the developer of a plugin to define a set of valid configurations that can be considered by the creation of all possible simulator configurations. We explicitly do not support the option to define several default values for each primitive plugin parameter individually as invalid combinations of default parameter values would have to be considered as well. In our opinion, this is more error-prone than defining all parameter value configurations individually. Besides, to prevent the `Adaptive Simulator` adapting primitive parameters that would change the results of a simulation run, e.g., the error parameter $\epsilon$ in $\tau$-leaping, a blacklist can be defined. Similarly, also for plugins a blacklist can be defined, e.g., in case it is known that some plugins might not be effective for a specific simulation experiment.

When executing an adaptation, the `Adaptive Simulator` exchanges its `internal-Simulator` completely, i.e., a new `Processor` is created based on the `ParameterBlock` of the chosen action. In JAMES II, this approach is easy to implement due to the strict separation of model and simulator. Thus, the newly created simulator can be initialized using the existing model object. Exchanging the internal simulator completely also relieves the `Adaptive Simulator` to be able to partly change the

current `internalSimulator` and no integrity checks have to be performed.

For the reward calculation, we developed a plugin type represented by the interface `IReward`. We have implemented two plugins for this type so far, one for computing the rate of simulated events per second and one for computing the logarithm of this rate:

$$R(\tau) = ln(\frac{e_\tau}{wct_\tau}) \tag{4.13}$$

where $wct_\tau$ is the duration to compute the trajectory $\tau$ in wall-clock time in seconds and $e_\tau$ is the number of simulation events that have been computed in this time span. For this, all base states must provide runtime information, e.g., easily calculated by the `Adaptive Simulator` in the `nextStep()` method:

```
//...
long before = System.currentTimeMillis();
internalSimulator.nextStep();
long after = System.currentTimeMillis() - before;
//...
```

The adaptation overhead is not considered by using this approach. Both metrics are simple to be calculated and they can also be used to calculate the reward of a subtrajectory of $\tau$. Measuring the runtime with `System.currentTimeMillis()` gives only a rough estimate of the real CPU time needed to compute the simulation step as it does not exclude the influence of other processes, threads, etc. For more precise estimates, the built-in class `java.lang.management.ThreadMXBean` could be used that provides more sophisticated methods to measure the CPU time of individual threads. However, for our purposes the more simple variant `System.currentTimeMillis()` provided sufficiently accurate results.

The next subsections illustrate in more detail the remaining implementation concepts of the `Adaptive Simulator`. First, we describe the *context* concept of JAMES II and how we use it to collect data for the base states (see Section 4.5.1). Next, the adaptation condition implementation (see Section 4.5.2) and the implementation of the learning and selection functionality (see Section 4.5.3) are explained.

### 4.5.1 Information Retrieval

There are three sources for data put into base states (see Figure 4.4): the model, the simulator, and the environment. Much environment information can be directly accessed by the `Adaptive Simulator` using built-in functions of Java, e.g., provided by `System`, `Runtime`, or `ThreadMXBean`. By using the class `System`, various system

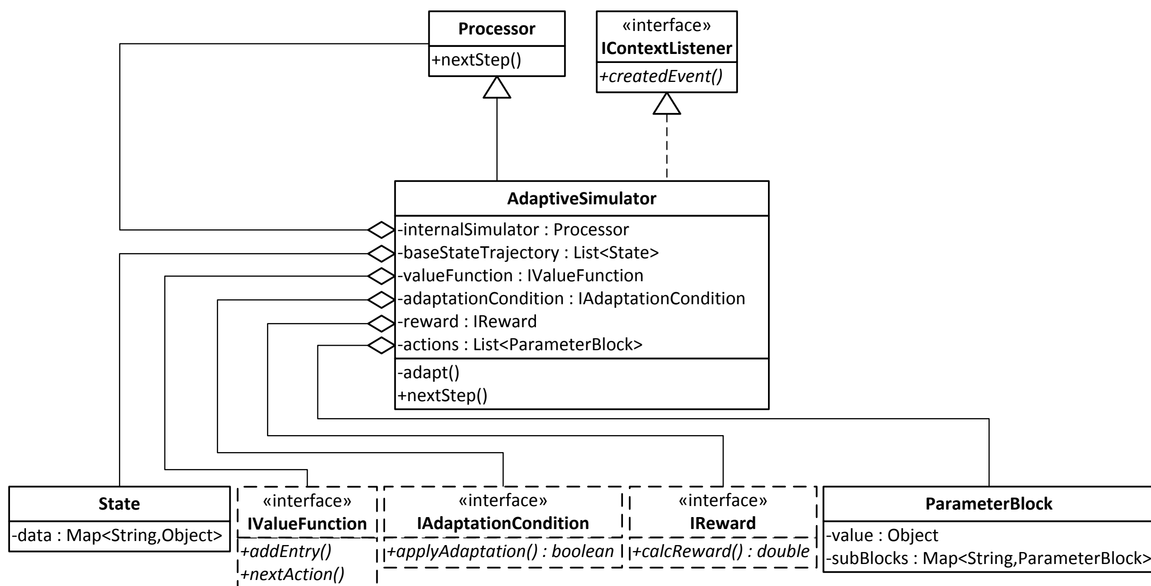



Figure 4.17: Overview of the `Adaptive Simulator` architecture. Interfaces with dashed borders define new JAMES II plugin types.

properties can be retrieved by the `getProperties(key)` method like the Java version (`"java.version"`), the Java vendor (`"java.vendor"`), the operating system name (`"os.name"`), and many more. The `Runtime` class can be used to retrieve information about the available memory in the JVM (`freeMemory()`), the number of available processors (`availableProcessors()`) etc. The `ThreadMXBean` class allows more specific queries, e.g., return the number of currently running threads (`getThreadCount()`), the total CPU time for a specific thread (`getThreadCPUTime(id)`), and much more information about the threads in the JVM.

Retrieving information from the model and the simulator is more difficult to be realized. In JAMES II, a model and a simulator can consist of a complex plugin hierarchy. A developer of a plugin must decide what information of this plugin could be worth to be considered for the `Adaptive Simulator`. If a plugin shall provide information, it has to implement the interface `IAdaptiveSource` with one method `getAdaptiveData():Map<String,Object>` returning this information. To get a list of all currently used plugins implementing this interface by an instance of the `Adaptive Simulator`, one could simply traverse all plugins by using Java reflection methods. Although this approach seems to be simple, it would be costly as the traverse would have to be done for every base state computation. Alternatively, one could create a list of the plugins implementing `IAdaptiveSource` once after an adaptation and reuse it for every base state computation. However, this approach would ignore all

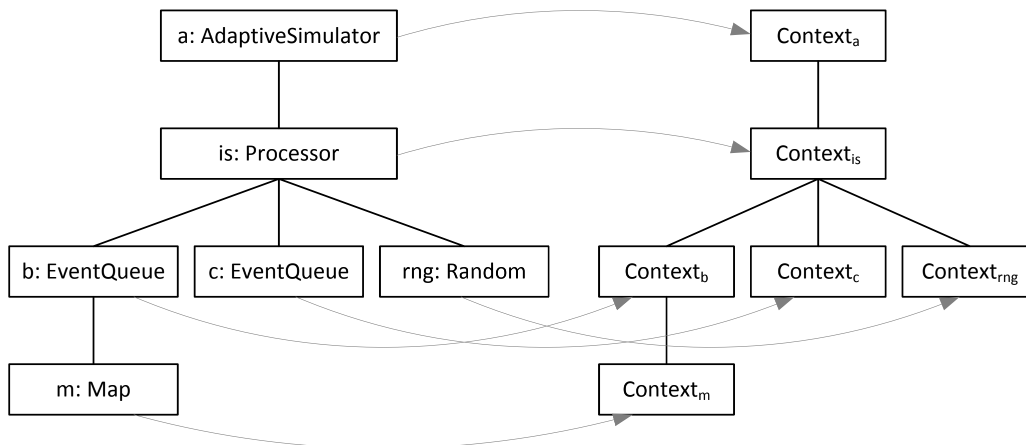



Figure 4.18: Illustration of a plugin hierarchy and its mirrored context hierarchy created automatically in JAMES II.

plugins created between two adaptations. Using the *context* concept of JAMES II allows realizing an elegant solution for these challenges. While creating a plugin and all of its sub plugins, i.e., a plugin hierarchy, JAMES II automatically also creates a mirrored context hierarchy, see Figure 4.18. This hierarchy explicitly represents the creation relation of all plugins, i.e., a context is related to the context it has been created in (its *parent* context) and to all contexts that have been created in it (its *child* contexts). Further, the `IContextListener` interface can be used to append listener to a context. Whenever a plugin is created, all listeners of its context and its ancestor contexts are notified about its creation. For example, if the plugin `c` in Figure 4.18 is creating a new plugin, all listeners of the contexts $\texttt{Context}_{\texttt{c}}$, $\texttt{Context}_{\texttt{is}}$, and $\texttt{Context}_{\texttt{a}}$ would be notified. Altogether, the *context* tree is suitable to enhance the computational reflection capabilities of JAMES II.

The `AdaptiveSimulator` class implements the `IContextListener` interface, see Figure 4.17 and it registers at its own context. Further, it registers at the context of the model plugin (the model context itself is not a child context of the simulator context as the model in JAMES II is not created by the simulator). In that way, it will always be notified whenever a new plugin is created within the model and within the internal simulator. When a notification is obtained, it is checked whether the created plugin implements the interface `IAdaptiveSource`. If so, it is added to the list of plugins that provide information for the base states. In the same way, the `Adaptive Simulator` is notified whenever a plugin is removed; it is then removed from the list of plugins providing information.

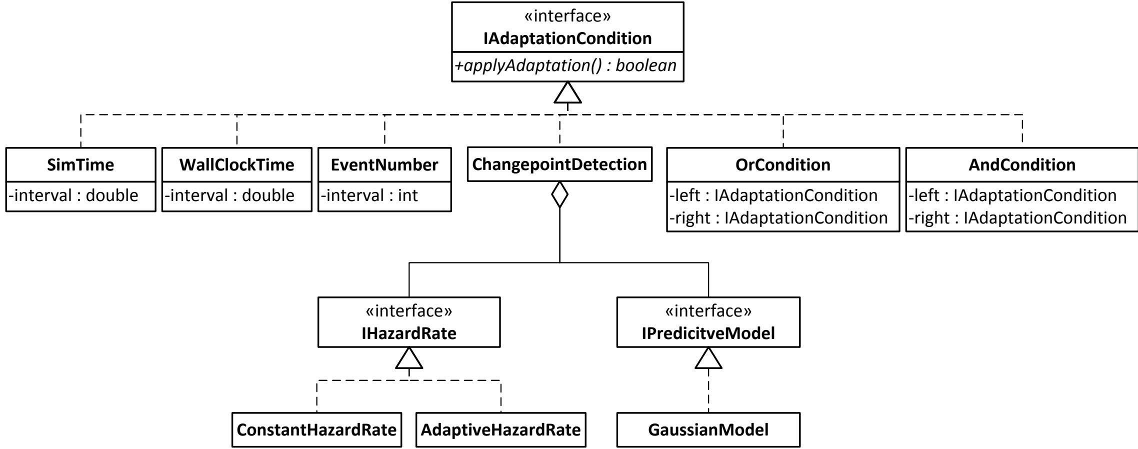



Figure 4.19: Class diagram of basic adaptation condition structure.

### 4.5.2 Adaptation Condition

For the adaptation conditions, we implemented simple conditions and the Bayesian changepoint detection algorithm presented in Section 4.4.1, see Figure 4.19. The simple conditions allow triggering an adaptation for a specific simulation time interval (`SimTime`), wall-clock time interval (`WallClockTime`), or executed event number interval (`EventNumber`). These conditions can be connected arbitrarily with the conjunction conditions `OrCondition` and `AndCondition`, e.g., to trigger an adaptation after 10 seconds wall-clock time and at least 100 executed events. The Bayesian changepoint detection algorithm is implemented in the class `ChangepointDetection`, mainly realizing the Algorithm 4.6. The computation of the hazard rate $h$ is done by separated plugins, i.e., `ConstantHazardRate` and `AdaptiveHazardRate`. The `ConstantHazardRate` uses a fixed value for $h \in [0, 1]$. The `AdaptiveHazardRate` considers the total number of executed events $e_{total}$ and the number of executed events since the last adaptation $e_{last}$ to refine the hazard rate $h$:

$$h' = h \cdot \frac{e_{total} - e_{last}}{e_{total}} + \frac{1}{e_{last}} \cdot \frac{e_{last}}{e_{total}} = \frac{h \cdot (e_{total} - e_{last}) + 1}{e_{total}}. \tag{4.14}$$

Besides these two simple hazard rates, more sophisticated methods, e.g., [200], could be added.

Finally, predictive probability distributions have to implement the interface `IPredictive-Model`. So far, we implemented one class `GaussianModel` that assumes the generative probability distribution to be a Normal distribution, as discussed in Section 4.4.1. However, since we separated the distribution from the rest of the changepoint detection

algorithm, it is straightforward to add further distributions to the model.

### 4.5.3 Value Function

Classes implementing the `IValueFunction` interface are organizing the state space for the `Adaptive Simulator` and realize the action selection mechanism, see Figure 4.20. The state space `IStateSpace` is defined by a set of states representing the macro state set $M$. So far, we implemented one state space that realizes a static regular grid (`Fixed`), one state space implementing the AVQ algorithm (`AdaptiveVectorQuantization`) and one state space implementing the DBPA (`DecisionBoundaryPartitioning`). These classes are responsible to map a state to its according cell. Further, they can split cells and merge cells if necessary. The same implementation of `ICellValue` is used for all cells in the state space, i.e., either all cells are of type `QValue` or all cells are of type `AValue`.

Basically, a `QValue` representing a macro state $m \in M$ contains for all actions $a \in A$ the q-values ($Q(m, a)$) and the selection counter ($N(m, a)$). The `QValue` class is used by `QLearning` that applies the basic Q-Learning rule, see Equation 4.1. Further, the `QLearning` class uses an implementation of the `ISelectionPolicy` like `EpsilonGreedy` to select an action for observed states. Thus, the selection policy is separated from the learning algorithm. We also implemented a policy `StaticSelection` that applies a predefined set of decisions. Such a policy is useful to reproduce simulation runs with the same adaptation sequence.

For the SASF, Ewald et al. already implemented various sophisticated multi-armed bandit policies using the `IMinBanditPolicy` interface. However, this interface cannot be combined with the `QLearning` class directly, because the implemented policies do not only select actions, but they also save and maintain the q-values their decisions are based on. To approximate the value of an action, they compute the mean of all received rewards for this action. The learning rule by using these policies can be interpreted as:

$$Q(s_t, a_t) = Q(s_t, a_t) + \frac{1}{N(s_t, a_t)} \cdot [r_{t+1} - Q(s_t, a_t)]. \tag{4.15}$$

This simplified version of the Q-Learning rule should still be useful for the `Adaptive Simulator`, because we also use $\alpha = \frac{1}{N[s_t, a_t]}$ for Q-Learning in `QLearning` as suggested by Sutton and Barto in [185] and we observed little impact of delayed rewards. The `AdaptiveSimulationRunner` that uses an `IMinBanditPolicy` for the decision making does not distinguish between different states of a simulation run, so that such a feature has not been considered during the development of this interface. Since we

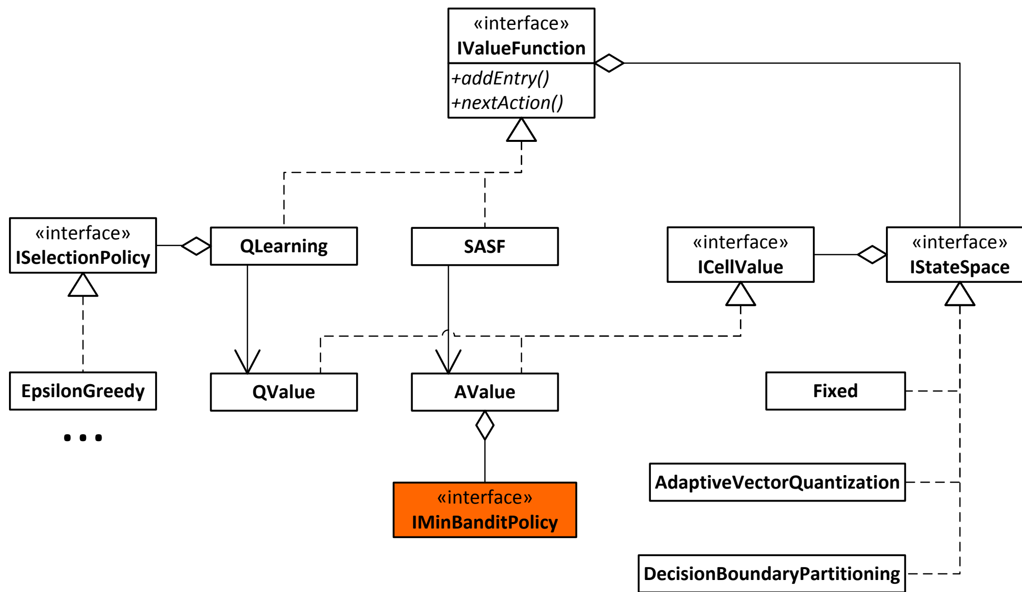



Figure 4.20: Class diagram of value function.

still wanted to use the existing policies, we created the class `SASF` implementing the `IValueFunction` interface and the corresponding cell value `AValue`. The name `SASF` emphasizes its purpose to *delegate* the learning and selection process to existing classes of the SASF. Each `AValue` contains one instance of an `IMinBanditPolicy` that manages these tasks. In the long term, Q-Learning and the multi-armed bandit policies of the SASF should be combined.

Besides the described properties, both `IValueFunction` implementations are implemented in a thread-safe manner, i.e., many `Adaptive Simulators` computing many simulation runs can use the same value function concurrently. If replications have to be executed, the same value function can thus be reused to improve the learning efficiency of the `Adaptive Simulator` for all replications. Furthermore, when used concurrently, updating the knowledge base becomes more complex, because states and rewards are observed concurrently. When a new state and reward is observed, the reward must be related to the previous state of the same simulation run. We solved this problem by using the *context* architecture of JAMES II again: when observing a state, not only the state itself, but also the context it was created in is saved that is used to associate the next reward with the correct state.

## 4.6 Measuring Adaptation Performance

We extend the concept of the *relative overhead* used to evaluate the `Adaptive-SimulationRunner` (see Section 2.5.3) to measure the effectiveness of the `Adaptive Simulator`. When applied to the `Adaptive Simulator`, we refer to this overhead as the **static regret** (following the term *regret* for MABPs). The performance of the `Adaptive Simulator` essentially depends on the quality of the q-values, i.e., how many explorations have been done to compute these values. To consider this issue, we sequentially execute several replications with the `Adaptive Simulator` using the same knowledge base, so that the q-values learned at the end of one replication are reused at the start of the next one, i.e., the `Adaptive Simulator` learns across all replications by successively improving its estimates of the true q-values. The static regret is the relative performance overhead of the `Adaptive Simulator` compared to a static selection of the best simulator after $n$ replications, see Equation 2.6 page 29.

However, since the performance overhead is compared with the best simulator, the static regret does not give any insights regarding the effectiveness of the adaptation mechanism of the `Adaptive Simulator`. To get this information, one has to estimate the performance of the best-performing `Adaptive Simulator` setting with an omniscient knowledge base. This can be done in two ways. First, one could use a promising configuration of the `Adaptive Simulator` and execute lots of replications to get a nearly "perfect" knowledge base. Afterward, a few additional replications could be executed by using this knowledge base. The performance while executing these additional replications could be used as a reference value for an "optimal" performing `Adaptive Simulator`. We followed this approach in [75]. The disadvantage of it is that the reference value essentially depends on the configuration of the `Adaptive Simulator`. Thus, it is not clear whether the reference value is really one of the best possible values or only an average estimate. In contrast to this approach, one could use performance values of all simulators to estimate the performance of an optimal `Adaptive Simulator`. For this, lots of replications have to be executed with every available simulator and configuration and the performance has to be measured frequently during the simulation, e.g., for each 100 executed simulation events. Next, the minimum observed performance value for each part of the replications is used to calculate the reference performance value for the optimal `Adaptive Simulator`. We applied this metric in [76]. The advantage of this approach is that the overhead of the `Adaptive Simulator` to access the knowledge base, to apply adaptations etc. is not considered in this reference value. Consequently, the reference value really represents the best possible result of an `Adaptive Simulator`. Nevertheless, the effort to calculate this value is higher compared to the first approach when the number of available simulators is large.

The performance estimate $reward^{opt}_{adaptive}$ of the best `Adaptive Simulator` can then be used to calculate the **dynamic regret** after $n$ replications

$$dr_n = \frac{\sum_{i=1}^{n} reward^i}{n \cdot reward^{opt}_{adaptive}}, \tag{4.16}$$

i.e., the overhead induced by exploration and applying adaptations compared to a "perfect" `Adaptive Simulator` always adapting to the best-performing simulator without any overhead.

## 4.7 Limitations and Open Challenges

Although the presented approach of the `Adaptive Simulator` is sophisticated and works in many scenarios, it has limitations. Most challenging are simulation runs that are executed in parallel using parallel discrete event simulation techniques. In principle, the `Adaptive Simulator` could be used as a central decision maker that changes the *global* algorithm to execute the simulation in parallel, e.g., change from a conservative to an optimistic simulation algorithm. Several complex challenges have to be solved in this case. For example, how to measure the performance of the total simulation? Further, the overhead to stop the simulation, to stop and adapt all logical processes might be higher than the benefit. Is it still worth to apply the `Adaptive Simulator`? Alternatively, the `Adaptive Simulator` could be applied to every logical process individually to reconfigure its parameters, e.g., to adapt the size of a time window. This approach sounds simple, but it must be guaranteed that the causality constraint is never violated, e.g., it should not be allowed to change from a conservative logical process to an optimistic logical process.

Moreover, the configuration of the `Adaptive Simulator` itself is another issue. The presented approaches and solutions emphasize that there are various possibilities to use the `Adaptive Simulator`, e.g., by using different state space generalization algorithms or adaptation conditions. Although we tried to use as few parameters as possible, this configuration problem has not been solved yet. One approach to solve it automatically would be to allow the `AdaptiveSimulationRunner` to select between several configurations of the `Adaptive Simulator`. It would eventually find the best performing setting automatically. However, this approach seems not to be suitable as the learning effort would be increased significantly. Meta-learning techniques [194] should be explored referring to the `Adaptive Simulator` to deal with this challenge.

Analogously to huge state spaces, huge action spaces cause various challenges. In adaptive software, huge action spaces typically occur if actions correspond to

configurations of components or algorithms, e.g., due to numerical or continuous parameters or due to the combinatorial opportunities of components. In this case, algorithm portfolios should be used to deal with this issue [93].

In general, no relationships between the performance values of simulators and different macro states are currently exploited. For example, if a simulator performs bad for almost all macro states, it seems not likely that it will perform better for the remaining macro states. It would also be interesting to use sensitivity analysis to identify similar performing simulators that can be summarized somehow.

Eventually, these ideas could be used to apply algorithm portfolios to the `Adaptive Simulator`. For example, a portfolio $P \subseteq A$ for the action set $A$ could be created incrementally in the following manner:

1. Initially, select $n \in \mathbb{N}$ actions randomly and add them to the portfolio, i.e., $|P| = n$.
2. After each $m \in \mathbb{N}$ simulation run executions, execute the following steps:
   (a) Remove all dominated actions from the portfolio $P$. An action $a_i$ is dominated by another action $a_j$ if $a_j$ performed almost always better than $a_i$.
   (b) Calculate a similarity graph $G = (P, E)$ for the actions, where an edge $e \in E$ means that both actions connected by $e$ have similar performance results.
   (c) Find a minimum vertex cover $MVC \subseteq P$ of this graph, i.e., a minimum number of actions so that each edge of $E$ is connected to at least one action in $MVC$.
   (d) Set $P := MVC$.
   (e) Finally, add randomly new actions to the portfolio so that $|P| = n$.

It has not been explored yet whether such an incremental creation of a portfolio would be effective for the `Adaptive Simulator`.

Besides, no mechanism to identify malicious plugins has been developed for the `Adaptive Simulator` yet. An approach to test simulator configurations used by the `Adaptive Simulator` is presented in the next Section 4.7.1.

### 4.7.1 Testing Component-based Stochastic Simulators

All options of the action set $A$ used by the `Adaptive Simulator` must compute valid results. Since the number of available options can be large, checking all options

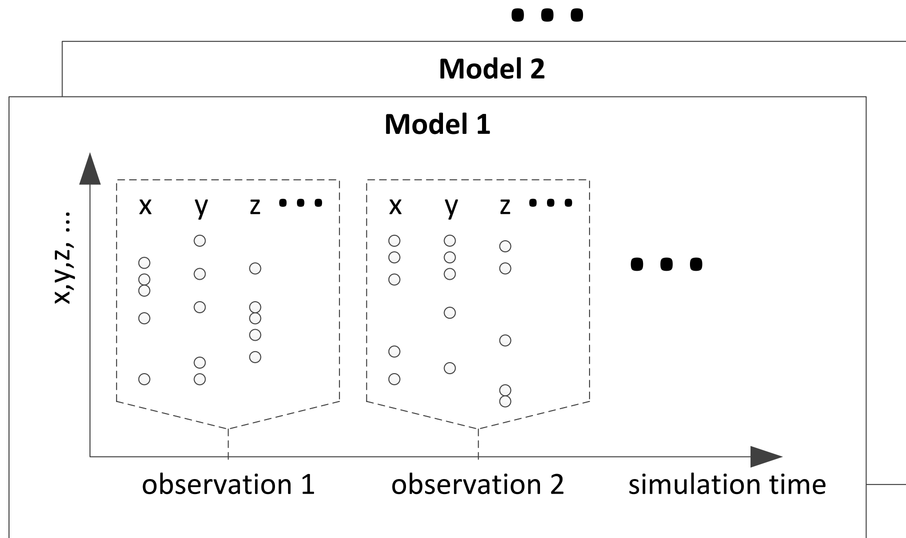



Figure 4.21: Per model and observation point, several values for each model variable are collected from the executed replications for each algorithm.

individually is challenging. For example, fractional factorial designs could be used to cover a certain level of interactions of plugins [104, p. 656ff.], e.g., two level interaction. Generally speaking, a two level interaction fractional factorial design can find errors that are caused either by individual plugins, or by the interaction of two plugins, i.e., if plugin `A` and `B` are selected together. Moreover, especially in discrete event simulation, simulation runs are often non-deterministic, so that an absolute decision whether an algorithm works well cannot be made. Further, whole trajectories have to be checked to validate the algorithms. The combination of both problems, i.e., a possibly huge number of options to test and non-deterministic simulation run trajectories, further complicates the testing task.

To deal with this problem, we developed a prototypical strategy [204]. The approach selects a subset of options $A_{sub}$ to be tested based on a simple heuristic: every plugin has to be used at least once within the subset of options. This heuristic reduces the set of options to be tested significantly, i.e., the plugin type with the largest number of plugins determines the number of options to be selected. Nevertheless, this method also makes it impossible to determine interaction errors of plugins.

For each selection tree in $A_{sub}$, simulation runs with several models are executed and several observations are made during each run, see Figure 4.21. Consequently, for each model, each observation point, and each model variable, an empirical distribution of values is collected for each selection tree. These empirical distributions are compared to results produced by a reference algorithm that we assume to be valid, i.e., without

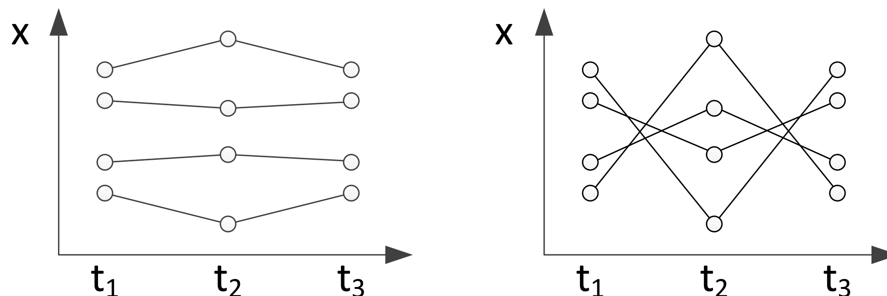



Figure 4.22: Although the left and the right trajectories have the same empirical distribution at the same time points, they are clearly different.

such a reference algorithm, the method cannot be applied. For the comparison, we use three statistical tests with a certain significance level $\alpha \in [0, 1]$: the Wilxocon rank-sum test [176, p. 513ff.], the Kolmogorov-Smirnov test [176, p. 577ff.], and the G-Test [180]. The null hypothesis is that a distribution of a model variable of the reference results and the according distribution of a selection tree from $\mathbb{A}_{sub}$ are sampled from the same population. Depending on $\alpha$, for each test one has to expect a number of type 1-errors, i.e., rejecting a true null hypothesis. Thus, our approach uses a binomial distribution with $p = \alpha$ to calculate the probability of the observed failures. If this probability is below a certain failure tolerance, the test fails. At least two tests must fail so that our approach concludes that the results are erroneous.

Although our approach has been often able to detect bugs in plugins, extending and improving the method is still an open research topic. First, the significance level $\alpha$ of the statistical tests and the statistical tests themselves have to be chosen carefully. If $\alpha$ is too high, too many tests will fail and a developer often has to unnecessarily check correct results manually. If $\alpha$ is too low, only obvious bugs are detected. Moreover, the approach currently does not distinguish between slight wrong results and dramatic wrong results, i.e., clear wrong results are not considered particularly. Further, due to the usage of the binomial distribution, a constant dramatic individual failure for a specific model and a specific observation point and a specific model variable would not make the approach to give alarm. The quality of the method also essentially depends on the chosen models and selected observation times. The approach does not consider properties of complete trajectories either, i.e., although the empirical distributions at the observation points might statistically represent the same population, the whole trajectories could still differ, see Figure 4.22. To address this issue, distance measurements for trajectories could be used, e.g., the Fréchet distance [41]. Alternatively, derivative trajectories could be calculated and compared. The approach currently does not exploit plugin information of algorithms to improve

the failure analysis. Also, a reference algorithm is needed to produce reference results, a strong assumption that will not always be fulfilled. Finally, it has not been adapted to test the `Adaptive Simulator` comprehensively. To test the results of the `Adaptive Simulator`, so far it has simply been added to $A_{sub}$. All these issues emphasize that the approach can and should be extended and improved in the future.

## 4.8 Summary

In this chapter, we developed the `Adaptive Simulator`— a generic simulator which supports *strong dynamic adaptations* and *compositional adaptations*. Thus, it can perform adaptations during the execution of a simulation run, adaptations can change the structure of the simulator, and it does not use a fixed set of adaptation options and a predefined adaptation strategy, but it uses reinforcement learning to learn autonomously which adaptations to perform. Consequently, it is not restricted to a particular modeling language or scenario.

The structure of the `Adaptive Simulator` considers the requirements identified in Section 4.1. We integrated the `Adaptive Simulator` in the modeling and simulation framework JAMES II, which is a suitable base for adaptive means as it supports key characteristics required by adaptive software (separation of concerns, component-based design, computational reflection), see Section 3. Thereby, we exploit the plugin system of JAMES II, i.e., the `Adaptive Simulator` itself is implemented in a component-based manner (see Section 4.5) and it uses selection trees (see Section 2.5.1) to represent all adaptation options, i.e., the simulator configurations, explicitly. By using the plugin system of JAMES II, the set of adaptation options can be calculated automatically. The `Adaptive Simulator` is realized as a wrapper by implementing the `Processor` interface all simulators in JAMES II must implement and can therefore be used transparently by the user. To execute the actual simulation run, the `Adaptive Simulator` uses an internal simulator that is exchanged completely when executing an adaptation. The adaptation process including observation, planning and executing an adaptation is executed separately from the simulation logic following the concept of a separated control loop, see Section 3.1. After an adaptation, the newly created internal simulator can simply use the current state of the model to initialize itself properly. There is no need to determine differences between the old and the new internal simulator and no data structures must be checked and updated to guarantee the integrity of the new internal simulator.

The `Adaptive Simulator` can collect information about the model, the simulator and the environment after each event execution to create *base states* ($\sigma \in \Sigma$) and *base state trajectories* ($\tau \in \Sigma^*$) that are aggregated to *aggregated states* ($s \in S$) at

the beginning of an adaptation process used by reinforcement learning. To deal with high-dimensional or infinite state spaces, we integrated three generalization methods into the `Adaptive Simulator`, see Section 4.3. The basic idea is to generalize an aggregated state $s \in S$ to a macro state $m \in M$, whereby $M \subseteq S \wedge |M| \ll |S|$. Each macro state represents an area of the state space — in the ideal case with a homogeneous performance behavior of all states represented by one macro state.

First, we developed a grid-based generalization method that applies a regular grid with a predefined grid size to generalize a concrete state. This method can be beneficial, but it is difficult to determine a suitable grid size and for different areas of the state space, different sizes might be effective. Second, we integrated the `Decision Boundary Partitioning Algorithm` (DBPA) [165] to the `Adaptive Simulator`, see Section 4.3.1, which is a dynamic generalization method adapting the generalization during runtime. This method uses hyperrectangular areas for the generalization and splits two adjacent areas, if the best action differ in both areas, the q-value of both best actions differ sufficiently, and both areas have been observed frequently. In the running example, we show that this method can be more efficient than a fixed grid, but its performance is not reliable and furthermore, also finding a suitable configuration producing at least some suitable results has been challenging. By integrating the `Adaptive Vector Quantization` (AVQ) [114] to the `Adaptive Simulator`, see Section 4.3.2, we found a more robust generalization algorithm for our application. This method uses a codebook consisting of a set of states called codewords and a nearest vector quantizer to map a state to its nearest codeword. The codewords therefore represent the macro states. The generalization of the state space is adapted during runtime by adding new codewords to the codebook or merging two codewords. Codewords are added to the codebook if the difference of two successive rewards within the same area is sufficiently high. Codewords are merged if the average pairwise difference of all rewards of two adjacent areas is sufficiently small. The running example shows that our version of the AVQ seems to be more robust than the DBPA, but the average results are worse compared to the best results of the DBPA.

To trigger adaptations, we pursued three concepts, see Section 4.4. First, a fixed adaptation condition based on the wallclock time, simulation time and number of processed events can be set. Fixed adaptation conditions can be useful, but it is challenging to define suitable conditions and different conditions might be effective for different phases of a simulation run. Second, we use a set of adaptation conditions and integrate them into the adaptation actions, i.e., an action does not only contain a simulator to be used, but also an adaptation condition to trigger the next adaptation. The `Adaptive Simulator` is therefore not only learning for which state to use which simulator, but also which adaptation trigger is most suitable. In some sense, this

approach refers to hierarchical reinforcement learning [10]. Although this approach enables the user to define a set of adaptation triggers, the action set increases with each trigger and therefore, the learning efficiency might be reduced. Third, we exploit the possibility to observe the performance of the `Adaptive Simulator` during runtime to apply changepoint detection algorithms to trigger adaptations. The assumption is that the computational requirements do not change in one simulation phase and therefore the event throughput of the simulator should be normally distributed. To apply this method, we couple the changepoint detection trigger with the exploration probability of the action selection policy to consider the need to gain performance knowledge. In general, we did not focus on model specific adaptation trigger, e.g., trigger adaptation after the execution of a rare event, since this would be application and model dependent and it would contradict the generality of the `Adaptive Simulator`.

In Section 4.5, we present important implementation aspects of the `Adaptive Simulator` and how the component-based architecture of JAMES II is exploited to realize the `Adaptive Simulator`. The `Adaptive Simulator` uses the context hierarchy reflecting the structure of a component to get all components providing information for it. Further, it is automatically notified if components of a simulator are removed or added to update its information listeners. Moreover, the available simulators including their configurations can be created automatically using the `Registry` of JAMES II. Finally, the `Adaptive Simulator` itself is realized as a component-based simulator — all important concerns (e.g., adaptation trigger and the reward function) of the `Adaptive Simulator` are separated into individual components making it flexible to integrate new methods and algorithms.

In Section 4.7, we also refer to open challenges and limitations of the `Adaptive Simulator`. For example, how to apply the `Adaptive Simulator` for parallel discrete event simulation or how to apply algorithm portfolios to deal with large action sets suitably. Further, it is not clear how to automatically test the validity of the `Adaptive Simulator` comprehensively — already testing the simulators applicable to a specific problem in case of stochastic simulation runs is challenging and not solved satisfactory yet, see Section 4.7.1.

Altogether, the four basic facets for adaptive software (see Section 3) related to the `Adaptive Simulator` can be described as follows:

- **Goals:** The runtime performance of simulation runs shall be improved. Thus, other measurements like memory consumption, accuracy of results, energy consumption etc. are not considered explicitly by the `Adaptive Simulator`.
- **Changes:** No model dependent and application dependent information is used to trigger an adaptation. Instead, conditions considering the wallclock time,

simulation time, or number of processed simulation events can be applied. Further, changepoint detection can be used considering the observed event throughput assuming a normally distributed throughput in one simulation phase.

- **Mechanisms:** The `Adaptive Simulator` realizes compositional adaptations. By using reinforcement learning, it learns autonomously how to adapt the internal simulator. It is not restricted to any specific simulator or modeling language.

- **Effects:** The internal simulator is exchanged completely by an adaptation. Since JAMES II requires developing well-defined model state objects, continuing the simulation run with a new simulator is simple to realize, i.e., no maintenance or integrity checks are needed after an adaptation.

After developing the concept of the `Adaptive Simulator`, we will evaluate its effectiveness and efficiency in the next Chapter 5 by conducting experiments with different modeling languages and simulators.

# Chapter 5

# Performance Experiments with the Adaptive Simulator

> A well-known problem in software engineering is that no matter what you do, user requirements will change.
>
> Raoul-Gabriel Urma et al. in Java 8 in Action

In Chapter 4, we designed the `Adaptive Simulator`, a generic adaptive simulator performing *strong dynamic adaptations* and *compositional adaptations*. It uses reinforcement learning to autonomously learn how to adapt during runtime and it is not restricted to a specific model or modeling language. In particular, we focused on different techniques for the `Adaptive Simulator` to deal with large and high-dimensional state spaces and different techniques for adaptation triggers. In this chapter, we evaluate the `Adaptive Simulator` referring to its components using simple benchmark models and complex models used in simulation studies. Most of the experiments are done using the modeling language ML-Rules— a modeling language for dynamically nested biochemical reaction networks [130], see Section 5.1. Due to its expressiveness, various computational challenges arise and therefore, it is a suitable language to develop component-based simulators and to apply the `Adaptive Simulator` to deal with the available options. To demonstrate the generality of the `Adaptive Simulator`, we apply it also to the modeling language SR (see Section 5.2.1) and to PDEVS (see Section 5.2.2).

Some parameters of the `Adaptive Simulator` are equal for all experiments. First, the reward is represented by the logarithmized ($base$ = 2) executed events per second. Second, delayed rewards are not considered, i.e., $\delta = 0$, and the learning rate $\alpha : \mathbb{N} \to [0,1]$is always set to $\alpha(N[s,a]) = \frac{1}{N[s,a]}$, see Equation 4.1 page 50.

Following the motivation of the performance metric *dynamic regret*, see Section 4.6,

to evaluate the effectiveness and efficiency of the `Adaptive Simulator`, we execute replications with the `Adaptive Simulator` using the same knowledge base, i.e., it reuses learned q-values at the start of a replication. However, to get reliable results, we also execute repetitions of whole simulation experiments.

In the following, we use the term *outer replication* to refer to one repetition of a simulation experiment. We use the term *inner replication* to refer to a replication of a simulation run within one simulation experiment. Analogously, we use the term *inner threads* to refer to the number of threads used to calculate the internal replications, and the term *outer threads* to refer to the number of threads to calculate the external replications. For example, setting the inner threads to 1 and the outer threads to 10 means that for each outer replication, the internal replications are executed sequentially, but 10 outer replications are executed in parallel.

To conduct the experiments, we exploit the experimental layer of JAMES II [87]. Originally, every experiment in JAMES II is executed by a central `BaseExperiment` object. For each BaseExperiment, various data can be set, e.g., a model to be simulated, model parameter configurations, replication criteria, a simulator to execute the replications etc. To realize the approach of inner and outer replications, we added a layer on top of the experimental layer. One `BaseExperiment` refers to one outer replication and the replications executed within a `BaseExperiment` object refer to the inner replications.

We used two computers to execute the experiments. All experiments except some experiments in Section 5.1.2.4 have been executed on a machine with the following configuration: *Intel(R) Core(TM) i7 CPU X990 @ 3.46 GHz* with activated Hyperthreading and deactivated TurboBoost, 24GB RAM, Windows 7, and Java 7. Due to the activated Hyperthreading, 12 threads can be executed in parallel. We always used 10 threads on this machine. Some experiments in Section 5.1.2.4 have been executed on another machine with the following configuration: *Intel(R) Xeon CPU X5690 @ 3.46 GHz* with actived Hyperthreading and deactivated TurboBoost, 48GB RAM, Windows 7 64bit, Java 8. On this machine, 24 threads can be executed in parallel. We always used 20 threads on this machine.

## 5.1 Experiments with ML-Rules

ML-Rules is a rule-based multi-level modeling language used to build dynamically nested cell-biological models [130, 129, 72]. The model entity types in ML-Rules are called species that can be attributed and dynamically nested. A multiset of species entities is called a solution. The dynamics of an ML-Rules model are described by rule schemes that define reactant patterns, products and a kinetic reaction rate. A reactant

pattern describes a species by its name (e.g., *A*, *Cell*, *Protein*), its attributes (e.g., *size*, *volume*, *age*), and further sub-species and their attributes etc. Since multiple levels can be considered by reactant patterns and products, ML-Rules explicitly supports downward and upward causation — an essential concept for many cell-biological systems [144]. ML-Rules is a complex and expressive modeling language, resulting in various computational challenges for a simulator of ML-Rules. Initially, in the following section 5.1.1, we present the basic features of ML-Rules and the consequences for the simulator. Afterward, we discuss experiment results achieved with ML-Rules and the `Adaptive Simulator` in Section 5.1.2.

### 5.1.1 Introduction

ML-Rules is a powerful modeling language supporting many complex features. The following examples extracted from [82] introduce the most important features step-by-step to illustrate ML-Rules' expressiveness. Besides, we have defined a formal semantics for ML-Rules [195, 81]. This formal semantics allows us to have a clear understanding of what is meant by a model and how it should be simulated. Further, a documentation describing the concrete syntax of ML-Rules as well as the concrete syntax formally defined in ANTLR4 syntax [151] is available in the ML-Rules source code repository (`https://git.informatik.uni-rostock.de/mosi/mlrules2`).

#### 5.1.1.1 Enzyme-Substrate-Product Model

Figure 5.1 shows an ML-Rules implementation of an *enzyme-substrate-product* network. Firstly, constants are defined (ll.2-3). For example, these constants can be used to calculate the initial amount of species or reaction rates. Afterward, species are defined (ll.6-10). Attributed species are not needed in this model. Attributes could have been defined within the parentheses, see Section 5.1.1.2. Next, the initial solution is defined (l.13), containing 1000 `E` entities and 1000 `S` entities.

Finally, the rule schemes of the model are defined (ll.16-22). Every rule scheme consists of three parts:

```
reactants -> products @ rate
```

For example, the first rule (l.16) describes the transformation of an enzyme (`E`) and a substrate (`S`) to a complex (`ES`). All rule schemes apply the law of mass action and therefore use reactant variables (`e` and `s` in the first rule) to access the amount of species via the `#` operator.

ML-Rules models are interpreted as continuous-time Markov chains and can be

```
1  // constants
2  k1: 1e-3; k2: 2; k3: 1; k4: 10; k5: 0.1;
3  n: 1000;
4
5  // species definitions
6  E(); // enzyme
7  S(); // substrat
8  ES(); // enzyme-substrat complex
9  P(); // product
10 EP(); // enzyme-product complex
11
12 // initial solution
13 >>INIT[n E + n S];
14
15 // rule schemes
16 E:e + S:s -> ES @ k1 * #e * #s;
17 ES:es -> E + S @ k2 * #es;
18
19 ES:es -> EP @ k3 * #es;
20
21 EP:ep -> E + P @ k4 * #ep;
22 E:e + P:p -> EP @ k5 * #e * #p;
```

Figure 5.1: From [82]. An *enzyme-substrat-product* model.

simulated by using the stochastic simulation algorithm [63], see Section 2.4. Following this interpretation, one state of an ML-Rules model corresponds to a well mixed solution encoded as a multiset of chemical species. Further, reaction rates are used firstly to calculate the probability to select a reaction to be fired and secondly to calculate the time advancement. Intuitively, the higher the rate of a reaction, the more likely it is to be executed. Referring to the *enzyme-substrate-product* model, the reaction set is constant with five reactions, i.e., every rule scheme can directly be mapped to one reaction.

#### 5.1.1.2 Attributed Species

Species can be equipped with attributes, e.g., to represent the age or the volume of an entity. Attributed species enable ML-Rules to support the concept of variables in reactant pattern. Figure 5.2 illustrates this concept with a simple cell cycle model. The only species `Cell` in this model has three attributes (l.5). The first attribute is a number, the second attribute is a string, and the third attribute is a boolean. Attributes do not have names in ML-Rules, but they are identified by their position

within the attribute tuple. The numbers of attributes of a species is fixed and cannot change. Consequently, all entities of one species have the same number of attributes. The meaning of an attribute is not an integral part of the ML-Rules model description, but can only be described informally in the documentation of the model, e.g., in form of comments in the model file. The attributes of the `Cell` shall be interpreted as follows:

- The first attribute represents the volume of a `Cell`.
- The second attribute represents the current state (`G1, SG2, M`) of a `Cell`.
- The third attribute represents a flag that shows whether the growth activity of a `Cell` is activated (`true`) or not (`false`).

The first rule scheme (ll.11-12) increases the volume of a `Cell` entity with activated growth activity. Two reactant variables `vol` and `state` are used. Therefore, the reactant `Cell(vol,state,true)` matches all `Cell` entities with an activated growth activity, independent of the concrete volume and state of the `Cell`. Given the initial solution, the simulator can therefore create one reaction based on this rule scheme with `vol = 1.0` and `state = 'G1'`. The reactant variables are reused in the product. The volume is increased by a random number calculated by calling `unif(0,1)`, i.e., the random number is sampled from a uniform distribution $U(0,1)$. Generally, expressions and functions can directly be used to calculate attribute values of products. The second rule scheme (ll.13-14) illustrates conditional rates, i.e., the rate of a concrete reaction is only greater than 0, if the volume of the considered `Cell` entity is greater than 2.0.

Simulating this model is more challenging compared to the introductory *enzyme-substrate-product* model. Initially, two reactions can be instantiated from the rule schemes:

```
Cell(1.0,'G1',true) -> Cell(1.0+unif(0,1),'G1',true) @ 0.1 * 10;
Cell(1.0,'G1',true) -> Cell(1.0+unif(0,1),'G1',false) @ 0.2 * 10;
```

The basic stochastic simulation algorithm can be applied using these two reactions, see Section 2.4. After firing one of the reactions, not only the propensities of both reactions have to be updated, but also new reactions have to be added to the reaction set. For example, when firing the second reaction, the resulting solution would be `9 Cell(1.0,'G1',true) + 1 Cell(1.0,'G1',false)` and the updated reaction set

```
1  // constants
2  k1: 0.1; k2: 0.5; k3: 0.4; k4: 0.3; k5: 0.2;
3
4  // species definitions
5  Cell(num,string,bool);
6
7  // initial solution
8  >>INIT[10 Cell(1.0,'G1',true)];
9
10 // rules
11 Cell(vol,state,true):c -> Cell(vol+unif(0,1),state,true)
12         @ k1 * #c;
13 Cell(vol,'G1',active):c -> Cell(vol,'SG2',active)
14         @ if (vol > 2.0) then k2 * #c else 0;
15 Cell(vol,'SG2',active):c -> Cell(vol,'M',active)
16         @ k3 * #c;
17 Cell(vol,'M',active):c -> 2 Cell(vol/2,'G1',active)
18         @ k4 * #c;
19 Cell(vol,state,active):c -> Cell(vol,state,!active)
20         @ k5 * #c;
```

Figure 5.2: From [82]. A simple *cell cycle* model.

would contain the following reactions:

```
Cell(1.0,'G1',true) -> Cell(1.0+unif(0,1),'G1',true) @ 0.1 * 9;
Cell(1.0,'G1',false) -> Cell(1.0+unif(0,1),'G1',false) @ 0.1 * 1;
Cell(1.0,'G1',true) -> Cell(1.0+unif(0,1),'G1',false) @ 0.2 * 9;
Cell(1.0,'G1',false) -> Cell(1.0+unif(0,1),'G1',true) @ 0.2 * 1;
```

In general, the reaction set of ML-Rules models is not fixed, but it can change frequently. Moreover, the complete reaction set of all possible species instantiations is usually infinite, e.g., due to an infinite number of different species entities, and therefore it cannot be calculated completely initially. Thus, it is often mandatory to regularly update the reaction set.

Altogether, two computational challenges can be identified. First, matching species must be found during runtime for reactants to calculate new reactions. To improve the efficiency of the matching process, filtering potential species efficiently by using sophisticated data structures might be suitable. Therefore, we implemented a plugin type for the ML-Rules simulator responsible for the species handling. Plugins of this type are mainly responsible for an efficient retrieval of species entities, e.g., to get all

entities with specific properties efficiently. We developed three plugins of this type:

1. `ListSpeciesHandling` saves all species entities in a list, i.e., the overhead of the maintenance is low, but the retrieval is not efficient. However, if the number of entities is small, this simple plugin might be the most efficient solution.

2. `MapSpeciesHandling` saves all species entities in a map indexed by species types. The overhead of this plugin is still low, but species can be retrieved efficiently by their type.

3. `GridSpeciesHandling` saves all species entities in a grid-file [143] that is a multi-key index data structure. By using this data structure, species can be efficiently retrieved by their attribute values, but the overhead of this data structure is relatively high.

A second computational challenge is the reaction set update. Analog to the idea of a dependency graph for the *Next Reaction Method* [62], in the ideal case, only invalid reactions are removed from the reaction set after a reaction execution and only new reactions are added to the reaction set. We added a flag `useDependencyGraph` to the ML-Rules simulator that activates or deactivates the usage of a dependency graph to update the rule instantiations, i.e., either the set of instantiations is cleared and calculated again completely after a reaction firing or only those instantiations are updated that are influenced by the fired reaction. Using a dependency graph can be advantageous if only few reactions are removed and added after each reaction execution, but it might be disadvantageous in case the reaction set changes significantly after each reaction execution.

#### 5.1.1.3 Compartments

ML-Rules explicitly supports dynamically nested entities, i.e., entities are able to contain other entities. If an entity contains other entities, it is called a *compartment*. All entities directly contained by an entity are called its *sub entities*. The entity that contains the sub entities is referred to as their *context*. The top-level context, which cannot changed by any reactions, is referred to as the *root context*. Further, in the first version of ML-Rules, see [130], all entities are treated in a population-based manner, i.e., all entities have an amount value representing the copy number of this concrete entity. Two entities are identical if they have a) the same context, b) the same species type, c) the same attributes and d) identical sub entities. For example, the solution

```
10 Organism[10 Cell[5 A + 5 B] + 5 Cell[4 A + 6 B]]        (5.1)
```

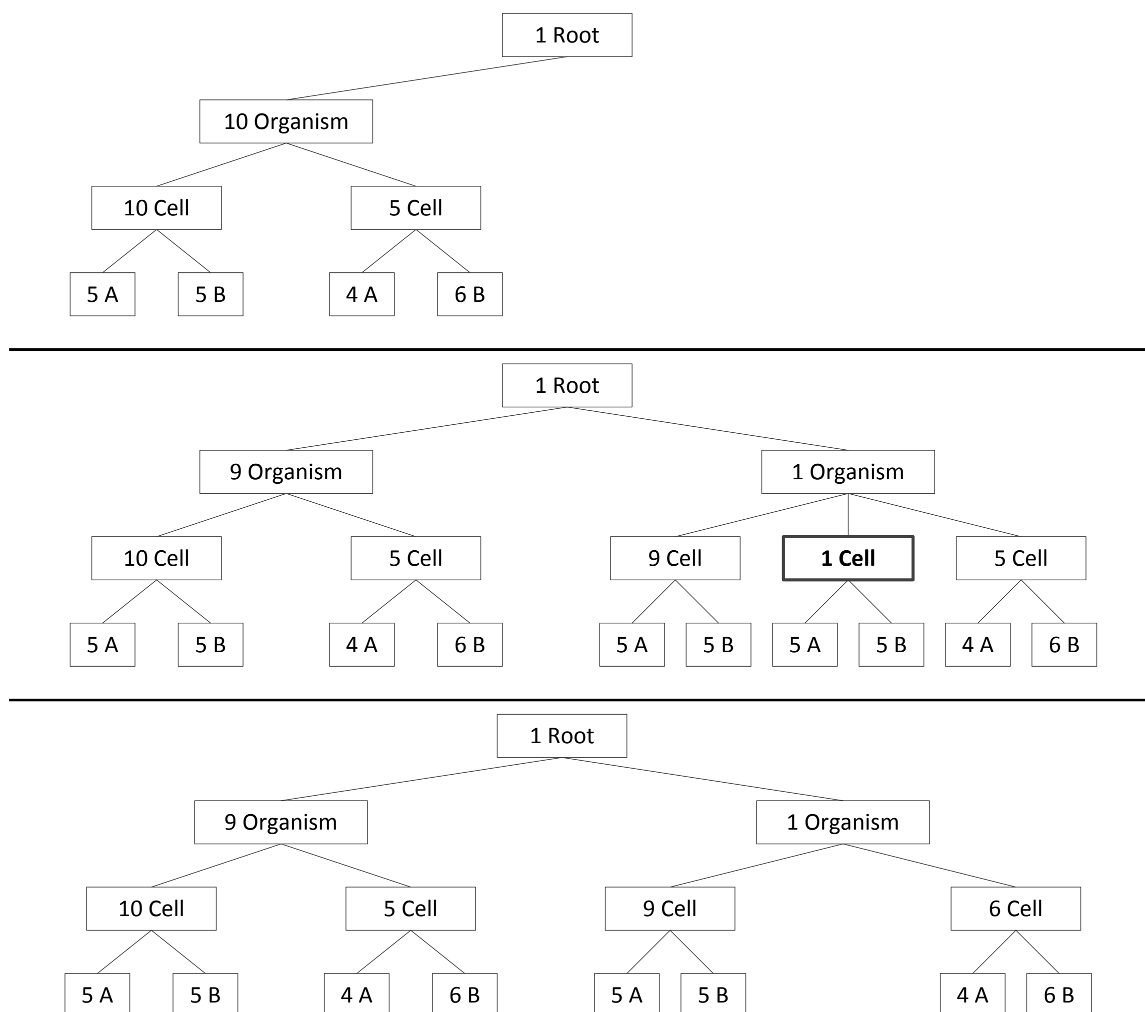



Figure 5.3: Top: Tree representation of the exemplary ML-Rules solution, see Equation 5.1. Middle: An individual branch has been extracted from the solution to execute rule $R_1$ in the marked context. Bottom: Result after executing `A -> B` in the marked context.

is one entity representing `10` identical `Organisms`, each containing `15 Cells`, whereby `10 Cells` containing `5 A` and `5 B` and `5 Cells` containing `5 A` and `3 B`. Figure 5.3 (top) shows a tree representation of this solution.

A population-based representation of compartments implies that the propensities of reactions must be multiplied with the copy numbers of their context hierarchy up to the root context. Considering the contexts is needed since an entity with a copy number $> 1$ represents a set of compartments, whereby the reactions inside this entity

can occur in each of these compartments. For example, suppose the rule scheme

```
A:a -> B @ #a;
```

is applied to the given exemplary solution, see Equation 5.1. The simulator can create two reactions based on this rule scheme, one within the context `10 Cell[5 A + 5 B]` ($R_1$) and one within the context `5 Cell[4 A + 6 B]` ($R_2$). To calculate the propensities of the reactions, the copy numbers of the contexts must be considered, i.e., the propensity of $R_1$ is $10 \cdot 10 \cdot 5 = 500$ and the propensity of $R_2$ is $10 \cdot 5 \cdot 4 = 200$.

If $R_1$ shall be executed next, an individual branch of the context hierarchy up to the root context must be extracted initially, see Figure 5.3 (middle). The extraction is necessary to restrict the reaction firing to an individual compartment hierarchy. If it would not be done and the amounts of `A` and `B` would be directly changed, *all* compartments represented by the context entity would be changed. After executing the reaction in the extracted context, changed compartments might be identical to other compartments and therefore, a merging procedure is processed successively from the context of the reaction to the root context. Figure 5.3 (bottom) shows the result of this process after the reaction execution.

Altogether, the extraction and the merging processes are expensive operations. However, population-based compartments can be beneficial since the rule scheme instantiation process does not have to be done for each individual compartment. Referring to the example, only two reactions are instantiated instead of one reaction per `Cell`, i.e., 150 reactions. Nevertheless, the solution shown in 5.1 is only typical for an initial solution of a model. With an increasing number of reaction firings, fewer and fewer compartments are usually identical, eventually resulting in an individual representation of compartments. In this case, the extraction and merging processes are additional costs for the simulator without any benefit.

Based on these observations, we changed the treatment of compartments in the current version of ML-Rules: compartments are always treated individually. Thus, the extraction and merging processes are avoided completely. Further, species types whose entities shall be used as compartments must be explicitly marked in the type definition. This enables the simulator to clearly distinguish between compartments treated individually and non-compartments treated in a population-based manner. Additionally, the integration of $\tau$-leaping is more simple with individual-based compartments, see Section 6.2.

Also compartments arise further challenges for the ML-Rules simulator. To improve the efficiency of the reaction instantiation and reaction execution, it might be beneficial to save sub species indexed by their type in a map. Therefore, we developed two plugins (`SpeciesWithList` and `SpeciesWithMap`) that use either a list to save the

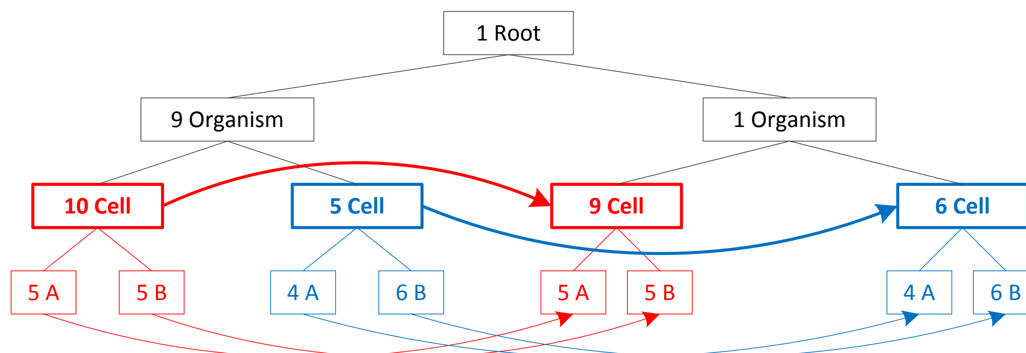



Figure 5.4: After executing `A -> B`, see Figure 5.3, a mapping of old entities to new entities with the same sub species can be created, i.e., reactions of the old contexts can be copied for the new contexts; only propensities must be updated.

sub species of a species or a map indexed by species type. Similarly, we developed two plugins (`SetReactionHandling` and `MapReactionHandling`) for the handling of the reactions. The plugin `MapReactionHandling` is indexing reactions by their context. Consequently, reactions can be removed efficiently. The plugin `SetReactionHandling` is either using a `HashSet` or an `ArrayList` to maintain the reactions.

Besides, to deal with population-based compartments more efficiently, we have developed two plugins for the reaction execution: `EqualsReactionExecution` and `IDReactionExecution`. These plugins do not influence the reaction execution itself, but they differ in how they deal with the extraction process of population-based compartments. The problem is that an extracted branch internally consists of newly created software objects, but all reactants of the selected reaction still refer to the software objects of the original branch. Therefore, a mapping has to be calculated for all reactants from the old branch to the new branch. Further, this mapping can be used to copy reactions from the old branch to the new branch for parts that have not been changed, i.e., not all rule scheme instantiations have to be calculated from scratch for the newly created contexts. For example, Figure 5.4 shows the result solution of the reaction execution illustrated in Figure 5.3 including a suitable mapping which can be used to copy reactions. All reactions that have the left red `10 Cell` entity as context can be copied accordingly for the right red `9 Cell` entity, since the sub species of both entities is identical. Only propensities of copied reactions have to be updated. Analog, all reactions that have the left blue `5 Cell` entity as context can be copied accordingly for the right blue `6 Cell` entity. Note that no mapping exists between the `9 Organism` context and the `1 Organism` context. Such a mapping would be invalid, since the sub species of both contexts are not identical and therefore, reactions might be invalid or new reactions might be possible in the new `1 Organism`

```
1  // constants
2  k1: 0.02;
3  k2: 0.15;
4  nBCatCell: 10000;
5  nBCatNuc: 4000;
6
7  // species definitions
8  Cell()[]; // compartment
9  Nucleus()[]; // compartment
10 BCat();
11
12 // initial solution
13 >>INIT[3 Cell[1 Nucleus[nBCatNuc BCat] + nBCatCell BCat]];
14
15 // rules
16 Nucleus[s?] + BCat:b -> Nucleus[BCat + s?] @ k1 * #b;
17 Nucleus[BCat:b + s?] -> Nucleus[s?] + BCat @ k2 * #b;
```

Figure 5.5: From [82]. A model of $\beta$-catenin proteins (`BCat`) shuttling into and out of the nucleus of a cell.

context. The plugin `EqualsReactionExecution` is determining the mapping by using the `equals()` method of the species software objects. This approach is simple to implement and does not need any additional auxiliary data structures. Nevertheless, the comparison of species entities can be complex in case of nested entities. The plugin `IDReactionExecution` is avoiding the direct comparison of entities by using an auxiliary `HashMap` maintaining the mapping between all original entities and copied entities.

#### 5.1.1.4 Multi-Level Rules

An essential feature of ML-Rules are dynamically nested entities and multi-level rules. For example, nested entities and multi-level rules can be used to describe diffusion proceses, as illustrated by the model in Figure 5.5. Compartments must be marked explicitly in the species definition by brackets after the attribute tuple, see lines 8-9. The initial solution of this model is a nested: three `Cell` compartments are created, each containing 10000 `Bcat` entities and one `Nucleus` compartment containing 4000 `Bcat` entities. Both rules of the model (ll.16-17) describe the shuttling of `Bcat` entities into and out of a `Nucleus` compartment. These rules use so-called *rest solution* variables (`<name>?`). These variables represent the whole content of the compartment matched to the reactant except entities that are already bound to other reactants.

```
1  // constants
2  k1: 0.001;
3  k2: 0.002;
4
5  // species definitions
6  Cell()[]; // compartment
7  Endo()[]; // compartment
8  Lyso()[]; // compartment
9  Particle();
10
11 // initial solution
12 >>INIT[100 Particle + 3 Cell[5 Lyso]];
13
14 // rules
15 Cell[s?] + Particle:p -> Cell[Endo[Particle] + s?] @ k1 * #p;
16 Endo[s1?] + Endo[s2?] -> Endo[s1? + s2?] @ k2;
17 Endo[s1?] + Lyso[s2?] -> Lyso[s1? + s2?] @ k2;
```

Figure 5.6: An abstract endocytosis model illustrating the creation and fusion of compartments.

Reactant variables are not used for compartments, as they are treated individually in ML-Rules.

Another model using compartments and multi-level rules is shown in Figure 5.6 representing an abstract endocytosis process. A particle (`Particle`) can enter a cell (`Cell`) engulfed by an endosome compartment (`Endo`) (l.15). Further, two endosomes can fuse (l.16) and an endosome can fuse with a lysosome (`Lyso`) (l.17). In contrast to the previous model, here the model structure is dynamic as endosome compartments are added and removed frequently.

In general, multi-level complicate the reaction instantiation process in ML-Rules, as the reactant matching must be extended to a recursive method. Further, in particular multi-level rules can be used to describe dynamic structures and consequently a dynamic reaction network. Thus, these rules also motivate an efficient update of the reaction network in ML-Rules. However, we have not developed specific plugins to deal explicitly with multi-level rules differently.

#### 5.1.1.5 Functions on Solutions

Besides the presented features, functions on solutions are an essential feature of the current version of ML-Rules. They had not been part of the initial ML-Rules version presented in [130]. These functions are able to deal with species and compartments

as input parameters. Based on the inputs, they can compute result species and compartments. For example, it is possible with functions on solutions to filter a solution, e.g., to remove all species of a specific type. Further, it is possible to compute statistics of a solution, e.g., calculate the average volume of all cells within this solution. Splitting a compartment is another typical application of functions on solutions. Altogether, ML-Rules provides various built-in functions, e.g., to count or filter solutions. The set of built-in functions is regularly extended. However, ML-Rules not only allows the modeler to use built-in functions, but we also added means to define own functions inspired by functional programming — in particular by the programming language Haskell [99].

Functions on solutions add another computational challenge to the ML-Rules simulator, e.g., complex functions must be interpreted frequently during runtime. Future work includes a transformation of functions on solutions to native Java code. This Java code then can be compiled at the beginning of simulation run to be executed more efficiently.

### 5.1.2 Experiments

To deal with different computational challenges of ML-Rules, we developed a plugin-based ML-Rules simulator [74]. Based on the features and components presented in the previous section, this ML-Rules simulator consists of the following components and parameters:

- Species Handling: `ListSpeciesHandling`, `MapSpeciesHandling`, and `GridSpeciesHandling`
- Reaction Handling: `SetReactionHandling` (with `HashSet` or `ArrayList`, `MapReactionHandling`
- Species Types: `SpeciesWithList` and `SpeciesWithMap`
- Reaction Execution: `EqualsReactionExecution` and `IDReactionExecution`
- `useDependencyGraph`: `true` or `false`

Consequently, for the experiments with the `Adaptive Simulator` and ML-Rules we can use $3 \cdot 3 \cdot 2 \cdot 2 \cdot 2 = 72$ simulator configurations. Note that none of these configurations change the semantics of the ML-Rules simulator.

The base state collected by the `Adaptive Simulator` for ML-Rules consists of four values: the number of different species $\sigma^s$, the number of removed and added species ($\sigma^s_+$ and $\sigma^s_-$) after each reaction execution, and the number of possible reactions $\sigma^r$.

Species in ML-Rules differ only when their types differ, they are nested in different species, their attributes differ, or their sub-species differ. Altogether, a base state $\sigma \in \Sigma$ comprises four attributes related to ML-Rules[1]:

$$\sigma = (\sigma^s, \sigma^r, \sigma^s_+, \sigma^s_-).$$

The function $p_1$ that aggregates a base state trajectory $\tau \in \Sigma^*$ to an aggregated state $s \in S$ is defined as follows:

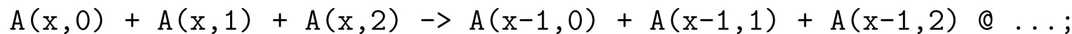
$$p_1(\tau) = s = \left( \frac{1}{|\tau|} \sum_{\sigma \in \tau} \sigma^s, \frac{1}{|\tau|} \sum_{\sigma \in \tau} \sigma^r, \frac{1}{|\tau|} \sum_{\sigma \in \tau} \frac{\sigma^s_+}{\sigma^s}, \frac{1}{|\tau|} \sum_{\sigma \in \tau} \frac{\sigma^s_-}{\sigma^s} \right), \tag{5.2}$$

i.e., the average species number, the average reactions number and the average ratio of added and removed species with respect to $\sigma^s$ are computed.

#### 5.1.2.1 Experiments with a Benchmark Model

For various experiments, we have developed a cyclic benchmark ML-Rules model with two alternating phases [75]. Figure 5.7 shows this model written in the current ML-Rules syntax.

Both phases describe degradation processes and need exactly 10,000 simulation events to be completed. The first phase degrades species that share an attribute value:



```
A(x,0) + A(x,1) + A(x,2) -> A(x-1,0) + A(x-1,1) + A(x-1,2) @ ...;
```

To compute all reactions based on this rule scheme, the simulator must determine all `A` species that have the same first attribute value. This can be done efficiently by using a multi-key data structure like the grid-file. The second phase of the model degrades species independently from all the others, i.e., no species that share attribute values have to be determined and a grid-file should not have benefits compared to a simple list. In contrast, the grid-file should perform worse due to its maintenance overhead.

For the first experiment we executed with the benchmark model, we only use two configurations of the ML-Rules simulator: one is using the plugin `ListSpeciesHandler` (*Simulator A*), and one is using the `GridSpeciesHandler` (*Simulator B*). The remaining plugins have been set as follows: `SpeciesWithMap`, `SetReactionHandling`, and `IDReactionExecution`. Further, the flag `useDependencyGraph` has been set to `true`. The average runtimes for each part of a simulation run with both simulators for the first $40,000$ simulation events is shown in Figure 5.8. Both phases of the model occur

[1]A base state also contains one meta-information, i.e., the wallclock time $t$ when the base state has been observed.

```
1  max : 100;
2
3  A(num,num);
4  B(num,num);
5  Switch(num);
6  System()[];
7
8  >>INIT [
9    1 System[
10     1 Switch(1) +
11     100 A(max, 0) +
12     100 A(max, 1) +
13     100 A(max, 2)
14   ]
15 ];
16
17 Switch(1) + A(x,0):t1 + A(x,1):t2 + A(x,2)
18   -> Switch(1) + A(x-1,0) + A(x-1,1) + A(x-1,2)
19   @ if (x>0) then #t1*#t2 else 0;
20 Switch(1) + 100 A(0,0) + 100 A(0,1) + 100 A(0,2)
21   -> Switch(2) + 50 A(max,0) + 50 A(max,1) + 50 A(max,2) + 50 A(max,3)
22   @ 1;
23
24 Switch(2) + A(x,i):t1
25   -> Switch(2) + A(x-2,i) + B(x,i) + B(x-1,i)
26   @ if (x>0) then #t1^(4) else 0;
27 System[Switch(2) + 50 A(0,0) + 50 A(0,1) + 50 A(0,2) + 50 A(0,3) + sol?]
28   -> System[Switch(1) + 100 A(max,0) + 100 A(max,1) + 100 A(max,2)]
29   @ 1;
```

Figure 5.7: The benchmark model from [75] written in the current ML-Rules syntax.

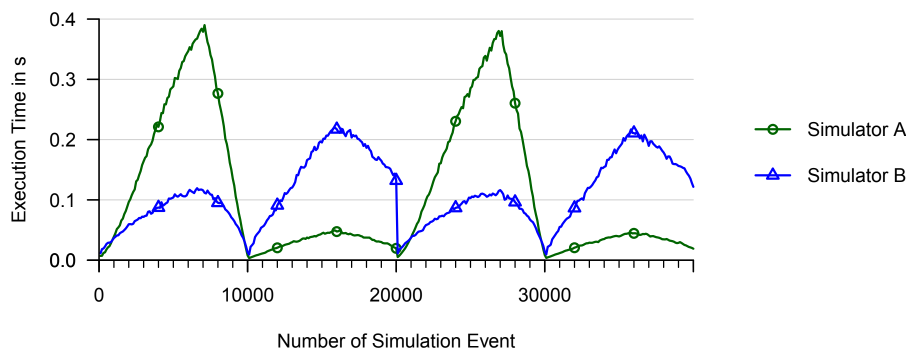



Figure 5.8: From [76]. Execution times for the two ML-Rules simulator configurations for the ML-Rules benchmark model. Each data point shows the execution time summed over 100 simulation events.

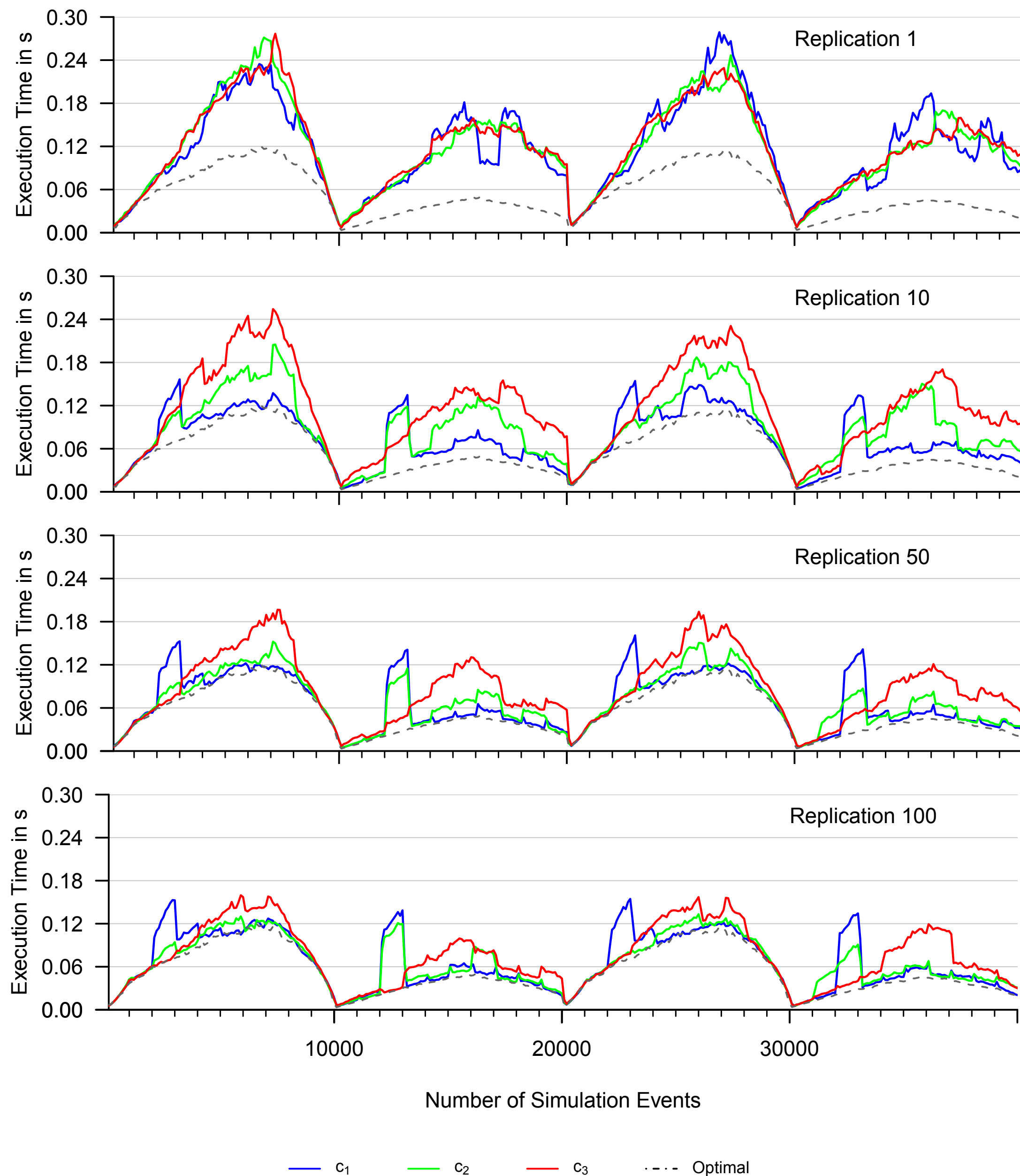



Figure 5.9: From [76]. In-detail view on averaged execution times during specific replications of the ML-Rules synthetic benchmark model, using the `Adaptive Simulator` with two actions and three different state space generalization configurations: $c_1$, $c_2$, and $c_3$. Each data point shows the execution time summed over 100 simulation events. The dotted lines denote the average execution times of the approximately optimal `Adaptive Simulator`.

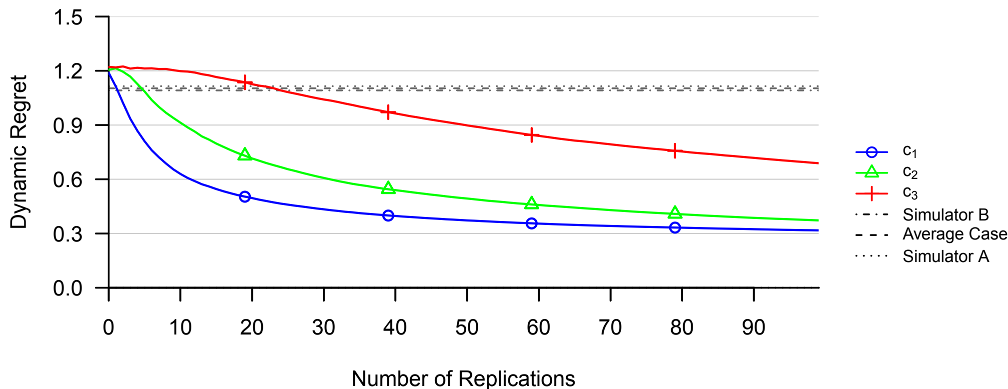



Figure 5.10: From [76]. The dynamic regret of the `Adaptive Simulator` on the ML-Rules synthetic benchmark model with three different static state space generalizations and two actions.

| | **Rep. 1** | **Rep. 10** | **Rep. 50** | **Rep. 100** |
|---|---|---|---|---|
| $c_1$ | 46.6 s (105%) | 29.2 s (66%) | 26.7 s (61%) | 26.8 s (61%) |
| $c_2$ | 46.9 s (106%) | 36 s (82%) | 27.5 s (63%) | 26.1 s (59%) |
| $c_3$ | 47.3 s (107%) | 45.7 s (104%) | 34.2 s (78%) | 29.6 s (67%) |

Table 5.1: From [76]. The average execution times of the `Adaptive Simulator` with the two-actions setup and the three used different static state space generalization configurations: $c_1$ (lowest granularity), $c_2$ (medium granularity), and $c_3$ (highest granularity). The relation to the average runtime of both simulator configurations ($\approx 44$ $s$) for each value is given in brackets.

twice within the first 40,000 simulation events. As expected, `Simulator B` that is using the grid-file to retrieve species is more efficient to simulate the first and third phase of the model (the third phase is a repetition of the first phase). Further, it performs worse than `Simulator A` for the second and fourth phase. On average, both simulators need $\approx 44$ $s$ to compute one simulation run of the benchmark model, i.e., they perform similarly. With optimal adaptations during runtime (use *Simulator A* during the second and fourth phase and use *Simulator B* during the first and third phase), it would be possible to achieve a runtime of $\approx 21$ $s$.

All in all, the model is suitable to demonstrate the effectiveness of the `Adaptive Simulator` and to analyze some of its properties. In [76], no dynamic state space mechanisms and no Bayesian changepoint detection have been available, so that we initially explored the impact of the static grid-based state space generalization with three granularities $c_i = (a, b, c, d)$: $c_1 = (50, 50, 0.5, 0.5)$, $c_2 = (10, 10, 0.1, 0.1)$, and $c_3 = (2, 2, 0.02, 0.02)$. An aggregated state $s = (w, x, y, z)$ is generalized by computing

the smallest multiple for each value of the used granularity that is smaller than the corresponding value of the aggregated state:

$$s = (a \cdot \lfloor \frac{w}{a} \rfloor, b \cdot \lfloor \frac{x}{b} \rfloor, c \cdot \lfloor \frac{y}{c} \rfloor, d \cdot \lfloor \frac{z}{d} \rfloor). \tag{5.3}$$

For example, the aggregated state $s = (7, 14, 0.43, 0.62)$ would be generalized to $(0, 0, 0, 0.5)$ with $c_1$, to $(0, 10, 0.4, 0.6)$ with $c_2$, and to $(6, 14, 0.42, 0.62)$ with $c_3$. Referring to the adaptation frequency, the `Adaptive Simulator` is regularly executing an adaptation every 1000 simulation events. Further, the `Adaptive Simulator` is using $\epsilon$-decreasing with $\epsilon = 5$ for the action selection. As motivated in Section 4.6, we executed 100 *inner replications* sequentially with the `Adaptive Simulator` and the same knowledge base, i.e., the number of *inner threads* is 1. The whole experiment has been repeated 50 times, i.e., the number of *outer replications* is 50. Figure 5.9 illustrates detailed runtime results of the `Adaptive Simulator`. Figure 5.10 shows the computed dynamic regret values and Table 5.1 gives an overview of the average runtimes per simulation run with the `Adaptive Simulator` and the three state space granularities. Clearly, in the long run the `Adaptive Simulator` performs better than both simulators. Further, the more fine-grained the state space, the more time is needed to learn efficient selection policies, i.e., the dynamic regret decreases slower the more fine-grained the generalization is chosen. However, a coarse-grained generalization can lead to permanent wrong decisions as can be seen in Figure 5.9, because aggregated states belonging to different phases of the simulation are mapped to the same macro state. Thus, the dynamic regret by using the granularities $c_1$ and $c_2$ converges to a value around 0.3.

In a second experiment with the benchmark model, we explored the effect of various selection policies. We use $\epsilon$-greedy ($\epsilon = 0.15$), $\epsilon$-decreasing ($\epsilon = 5$), *Upper Confidence Bound (UCB1)* [6], *Interval Estimation* ($\alpha = 0.05$) [101], and SoftMax [193]. These policies work as follows. The $\epsilon$-greedy policy chooses — based on its current knowledge — the best action $a$ with the constant probability $p(s, a) = 1 - \epsilon$ and otherwise a random action. The $\epsilon$-decreasing policy couples the probability $p(a)$ to select the best action $a$ on the number $n$ of selections: $p(a) = 1 - \min(1, \frac{\epsilon}{n})$. The policy *UCB1* initially chooses each action once. Afterward, for each following decision, the action which maximizes

$$\hat{\mu}_i + \sqrt{\frac{2 \ln n}{n_i}} \tag{5.4}$$

is chosen, where $\hat{\mu}_i$ represents the current estimate of the performance of action $a_i$, $n$ the number of occurrences of the according state $s$, and $n_i$ the number of selections of $a_i$ after observing $s$. The second summand converges to zero over time so that the exploration rate converges to zero. Consequently, *UCB1* is a zero-regret policy.

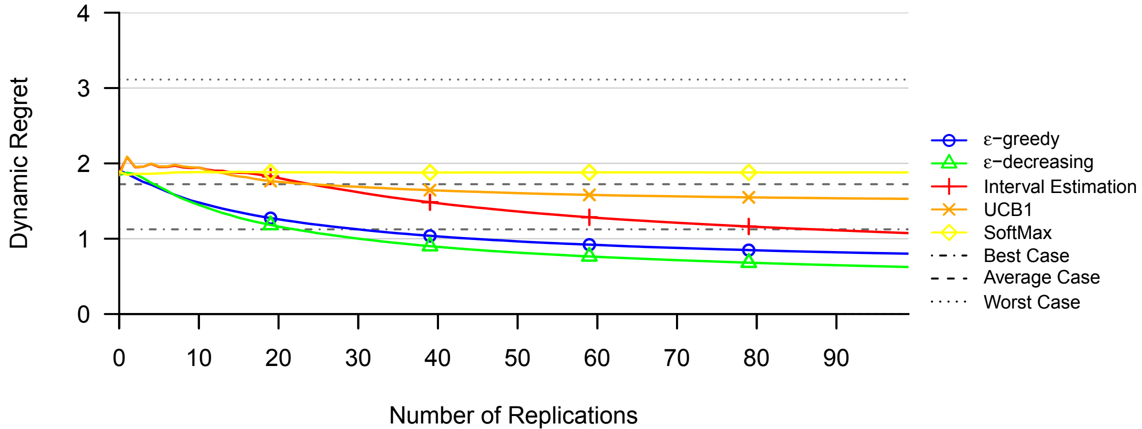



Figure 5.11: From [76]. The dynamic regret of the `Adaptive Simulator` setups on the ML-Rules synthetic benchmark model with five different action selection policies and 36 actions.

Similar to *UCB1*, the *Interval Estimation* policy initially chooses each action once. Afterward, confidence intervals of the performance for each action are computed. For each decision, the action with the highest upper bound of its confidence interval is chosen. Finally, the policy *SoftMax* computes the probability $p_i$ to choose action $a_i$ with

$$p_i = \frac{e^{\hat{\mu}_i/\tau}}{\sum_{j=1}^{|A|} e^{\hat{\mu}_j/\tau}}, \tag{5.5}$$

where $\tau \in \mathbb{R}^+$ is the temperature parameter chosen by the user and $\hat{\mu}_i$ is again the current estimate of the performance of the action $a_i$. The higher $\tau$ is chosen, the more exploration takes place. This strategy does not select each action once initially, but it is not a zero-regret policy.

For this experiment, we allow the `Adaptive Simulator` to choose between 36 simulator configurations (all plugins are used and always `useDependencyGraph = true`) of the plugin-based ML-Rules simulator. Again, we executed 100 inner replications and 50 outer replications and we used 1 inner thread and 10 outer threads. For the state generalization, we use the granularity $c_2$. Figure 5.11 illustrates the dynamic regret trajectories of the policies. As expected, the dynamic regrets of $\epsilon$-greedy and $\epsilon$-decreasing decrease quickly as they use their knowledge rather directly. The results match the results by Ewald et al. observed in [47], i.e., $\epsilon$-greedy and $\epsilon$-decreasing perform better compared to the other policies and although $\epsilon$-greedy performs better than $\epsilon$-decreasing initially, $\epsilon$-decreasing outperforms it after a few replications.

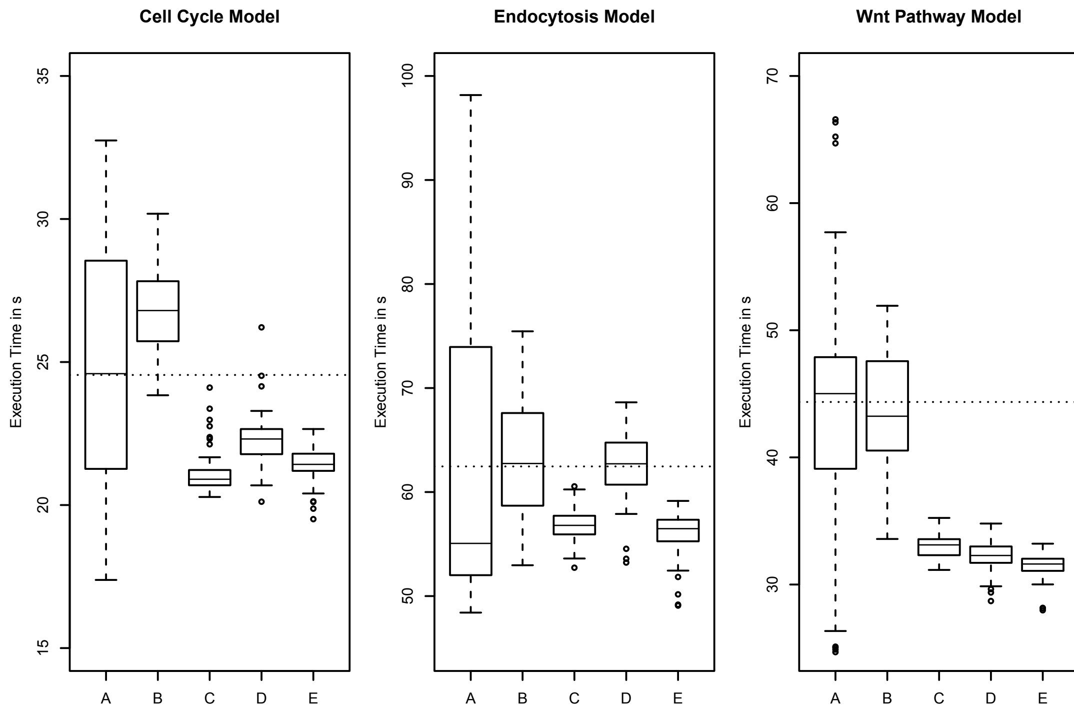



Figure 5.12: From [76]. Boxplots of the average execution times to compute one replication with the 36 ML-Rules simulator configurations (A), the `AdaptiveSimulationRunner` executing ten replications (B), the `AdaptiveSimulationRunner` executing 100 replications (C), the `Adaptive Simulator` executing ten replications (D), and the `Adaptive Simulator` executing 100 replications (E). The horizontal dotted lines represent the execution time per replication for the corresponding model, averaged over all 36 simulator configurations.

#### 5.1.2.2 Experiments with Complex Models

Besides the experiments done with the ML-Rules benchmark model, we have executed further experiments with more complex models used for simulation studies: a Cell Cycle model [130], an Endocytosis model [74], and a Wnt/$\beta$-catenin pathway model [131, 132, 69], see Appendix A.1, A.2, and A.3 for the ML-Rules implementations of these models. For initial experiments, we use again 36 simulator configurations (all plugins are used and always `useDependencyGraph = true`), the static grid-based state space generalization $c_4 = (5, 5, 0.1, 0.1)$, the $\epsilon$-decreasing ($\epsilon = 5$) policy, and an adaptation execution every 1000 simulation events. Additionally, we executed experiments with the `AdaptiveSimulationRunner` (see Section 2.5.3) using $\epsilon$-decreasing ($\epsilon = 5$). Here, we executed 100 inner replications with 10 inner threads — the Cell Cycle and the Endocytosis model for $100,000$ simulation events per replication, the

| | **Cell Cycle** | **Endocytosis** | **Wnt Pathway** |
|---|---|---|---|
| 36 SA configurations | 24.5 s ($\sigma = 4.4$ s) | 60.9 s ($\sigma = 13$ s) | 44 s ($\sigma = 12$ s) |
| `Adaptive Simulator` executing ten replications | 22.2 s ($\sigma = 0.9$ s) | 62.1 s ($\sigma = 3.4$ s) | 32.3 s ($\sigma = 1.2$ s) |
| `Adaptive Simulator` executing 100 replications | 21.3 s ($\sigma = 0.7$ s) | 55.9 s ($\sigma = 2.2$ s) | 31 s ($\sigma = 1.15$ s) |
| `AdaptiveSimulationRunner` executing ten replications | 26.9 s ($\sigma = 1.8$ s) | 63.3 s ($\sigma = 5.8$ s) | 43.7 s ($\sigma = 4.5$ s) |
| `AdaptiveSimulationRunner` executing 100 replications | 21.8 s ($\sigma = 0.8$ s) | 56.9 s ($\sigma = 1.5$ s) | 33 s ($\sigma = 0.9$ s) |

Table 5.2: From [76]. Average execution times for one replication of the corresponding model, for the 36 SA configurations, the `Adaptive Simulator`, and the `AdaptiveSimulationRunner`.

Wnt/$\beta$-catenin pathway model for 500,000 simulation events per replication. Further, we executed 50 outer replications per model with 1 outer thread. Figure 5.12 illustrates the averaged runtime boxplots of the different simulators. Both adaptation techniques, the `AdaptiveSimulationRunner` and the `Adaptive Simulator`, outperform the average performance of the simulator configurations when executing 100 replications. For the Cell Cycle model and the Wnt/$\beta$-catenin pathway model, the `Adaptive Simulator` already outperforms the average performance after ten replications. Furthermore, the results of the `Adaptive Simulator` are rather robust, especially the worst-case performance is much better than that of the worst simulator configuration. Compared to the `AdaptiveSimulationRunner`, the `Adaptive Simulator` performs similarly when executing 100 replications, but much better when executing ten replications. It works better because it directly uses its learned knowledge for the initial ten replications, whereas the `AdaptiveSimulationRunner` has to choose the simulator configurations for the first ten replications randomly. Interestingly, the `Adaptive Simulator` never performs better than the best simulator configurations.

#### 5.1.2.3 Changepoint Detection Experiment

In ML-Rules, we analyzed the effectiveness of our Bayesian changepoint detection algorithm for the adaptation condition, see Section 4.4.1, by executing simulation runs with the Wnt/$\beta$-catenin pathway model [80]. We executed 100 inner replications sequentially, i.e., with 1 inner thread and we executed 50 outer replications with 10 outer threads. Each replication has been executed for 100,000 simulation events. We set $a_{min} = 100$, because the needed execution time for one simulation event is typically smaller than 1 $ms$. For the action selection, we set $\epsilon$-decreasing with $\epsilon = 5$. For

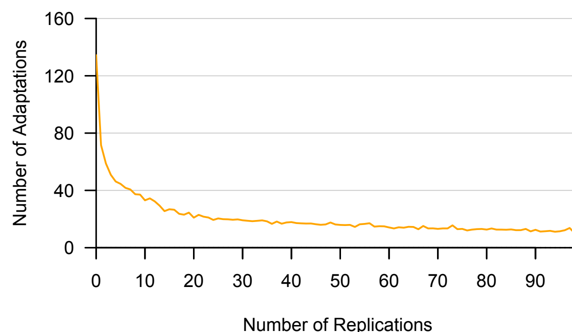



Figure 5.13: From [80]. Due to many unknown states and high exploration rates of the $\epsilon$-decreasing strategy at the beginning of the experiment, many adaptations are executed per replication due to enforced adaptations. As the experiment continues, the number of adaptations constantly decreases to approximately ten adaptations per replication.

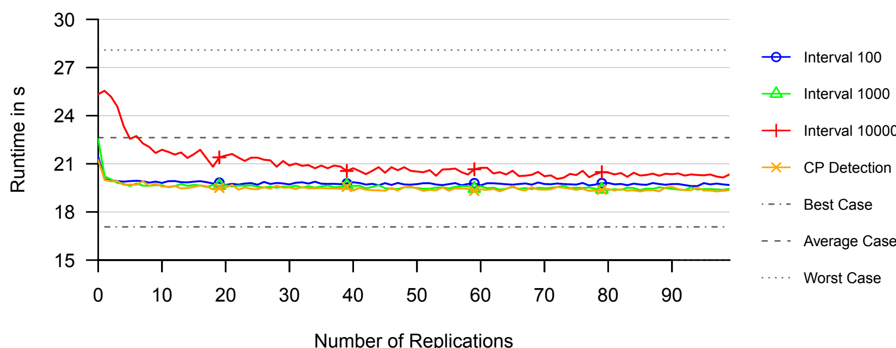



Figure 5.14: From [80]. Performance results of the Wnt/$\beta$-catenin pathway model experiment with different adaptation intervals and the changepoint detection algorithm.

the state space generalization, we use the static grid-based generalization with the granularity $c_4 = (5, 5, 0.1, 0.1)$. For the action set, we have not used all 36 simulator configurations for the experiment, but only 12 options constructed with the plugin sets {`MapSpeciesHandler`, `GridSpeciesHandler`}, {`EqualsReactionExecution`, `IDReactionExecution`}, and {`SetReactionHandling` with `HashSet` and `ArrayList`, `MapReactionHandling`} and `useDependencyGraph` = `true`.

We executed the experiment with three static adaptation intervals (100, 1000, 10000) and with four different configurations of the changepoint detection algorithm with $\delta \in \{10, 100\}$ and $h \in \{0.01, 0.0001\}$. Averaged performance results are illustrated in Figure 5.14. Only one result with the changepoint detection algorithm is shown, because the results have been very similar with all four configurations, i.e., the impact of $\delta$ and $h$ have been low. This emphasizes that the algorithm seems to be robust

and default parameter values might be suitable for most cases. Independent from the concrete adaptation condition, the `Adaptive Simulator` outperforms the average performance of the twelve simulator configurations. When executing adaptations rarely, i.e., every 10000 simulation events, the learning efficiency reduces. In ML-Rules, the costs of an adaptation are low, so that it seems to be suitable to compute many adaptations: the performance with adaptations every 100 simulation events only worsen slightly due to the adaptation overhead. The changepoint detection algorithm as well as a static adaptation interval 1000 produce the best results of the `Adaptive Simulator`. However, the changepoint detection algorithm needs much fewer adaptations to achieve this result, see Figure 5.13. Although this observation is not important for ML-Rules, it becomes important when adaptations are costly, see Section 5.13.

#### 5.1.2.4 Dynamic State Space Generalization Experiment

Continuing the running example from Section 4.3, Figure 5.15 shows the dynamic regret of the used state space generalizations for the ML-Rules benchmark model. The results show that a static grid generalization with a suitable grid size can be efficient initially, but it will be outperformed by good dynamic generalization methods in the long run. Further, since a good grid size is typically not obvious, one cannot expect to achieve good results with a static grid generalization without spending effort to analyze the concrete application scenario. Finally, although the DBPA performs best after 100 replications, the dynamic regret is misleading here. For all used generalizations except the DBPA, the variance of the performance results is small, so that the dynamic regret nicely reflects their performance behavior one can expect when executing one simulation experiment with the `Adaptive Simulator`, see Figure 4.15 page 72. For the DBPA, however, the variance is much higher and therefore, when executing one simulation experiment with the `Adaptive Simulator` and the DBPA, the results could be much better compared to the dynamic regret of the DBPA, but also much worse.

Besides, based on the experiment described in the previous Section 5.1.2.3, we repeated the experiment with the AVQ-based dynamic state space representation, see Section 4.3.2. Thus, we simulated the Wnt/$\beta$-catenin pathway model for $100,000$ simulation events. We executed again 100 inner replications sequentially and repeated the overall experiment 50 times, i.e., we executed 50 outer replications. For the adaptation condition, we use the changepoint detection algorithm developed in [76] with default parameters ($\delta = 10$, $h = 0.01$). Moreover, again we use $\epsilon$-decreasing with $\epsilon = 5$ for the action selection policy. The minimum reward difference to add new codewords to the codebook $\alpha$ has been set to $\alpha = log_2(1.5) \approx 0.585$. Since

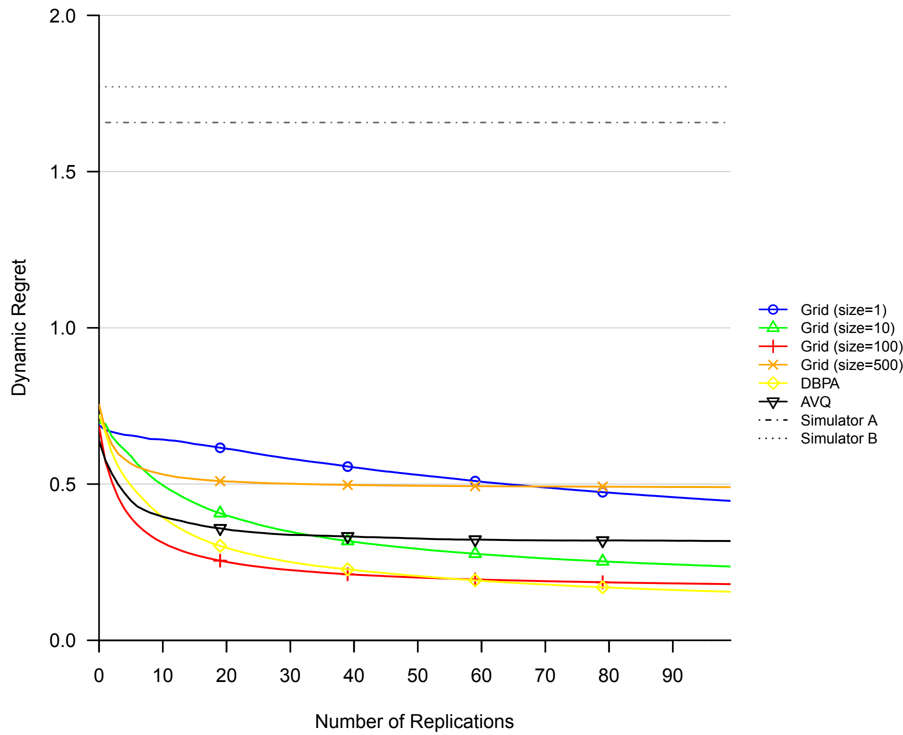



Figure 5.15: Dynamic regret for the ML-Rules benchmark model and different state space generalizations.

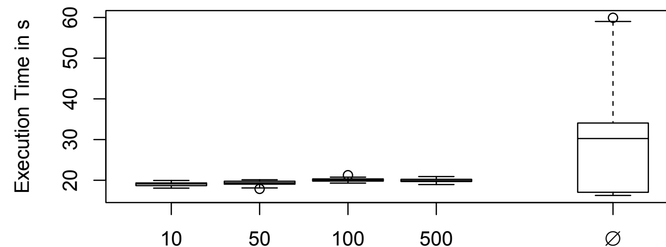



Figure 5.16: From [79]. Average execution time distributions of the Wnt/$\beta$-catenin pathway model with the `Adaptive Simulator` using the changed AVQ with $m \in \{10, 50, 100, 500\}$. The right distribution illustrates the average execution time distribution of the available 24 simulator configurations.

the logarithmic event throughput is used as reward, a throughput difference of at least 50% must be observed to add a new codeword to the codebook. Accordingly, we set $\rho = \alpha^2 \approx 0.343$. Here, we use 24 simulator configurations for the action set, constructed with the plugin sets {`ListSpeciesHandler`, `MapSpeciesHandler`,

GridSpeciesHandler}, {EqualsReactionExecution, IDReactionExecution}, and {SetReactionHandling with HashSet, MapReactionHandling} and the flag use-DependencyGraph $\in$ {true, false}. Figure 5.16 illustrates the runtime results of the Adaptive Simulator with different values of macro states sizes $m \in \{10, 50, 100, 500\}$ compared to the runtime distribution of the used 24 simulator configurations. Since the merging process of our variant of the AVQ based state space algorithm is executed after each replication execution, 99 merging processes were executed during the execution of 100 replications (no merging before the first replication and after the last replication). The Adaptive Simulator achieves almost the performance of the best simulator configuration with all values of $m$ and the performance distribution is clearly better compared to the performance distribution of the 24 simulator configurations that basically represents the distribution of a "random choice". A few macro states is sufficient to achieve good results. Further, the algorithm seems to be robust since the performance only worsen slightly with a higher number of macro states owing to a higher exploration effort.

## 5.2 Other Modeling Formalisms

Besides experiments with ML-Rules, we tested the Adaptive Simulator with two other modeling formalisms available in JAMES II: SR and DEVS. The following sections give a short overview about the results the Adaptive Simulator has achieved referring to these formalisms.

### 5.2.1 Species-Reactions (SR)

An important data structure in discrete event simulation are event queues that maintain the queue of all events to be processed [84, p. 142ff.]. This maintenance is challenging in many ways, e.g., new events are stored frequently into the event queue, events are regularly removed from the queue, and the event with the minimum time stamp shall always be accessed efficiently. Since the importance of these facets and thus the performance of an event queue depend on the concrete model to be executed [88], several event queue implementations exist to suit specific characteristics. For example, bucket-based queues like the calendar queue [19] partition all events based on time stamps or based on time periods. An alternative is the MList [65]. This queue uses three data structures to organize events. Events with a time stamp near the current simulation time ($t + \epsilon$) are stored in a sorted list. Events with a time stamp larger than $t + \epsilon$ but smaller than $t + \delta$ ($\delta > \epsilon$) are stored in a bucket-based queue. All remaining events are stored in a third unsorted list. A performance evaluation of

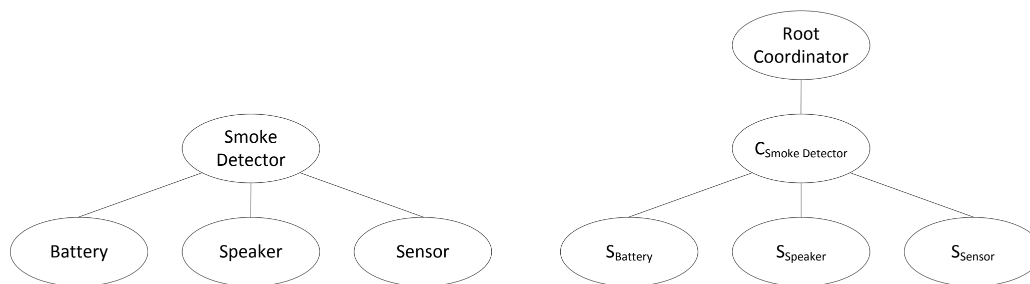



Figure 5.17: Model tree (left) and processor tree (right) of a simple smoke detector model.

various event queues is shown in [88, 97].

Like ML-Rules, Species-Reactions (SR) is a rule-based modeling formalism to describe biochemical reaction networks. It is also available within the modeling and simulation framework JAMES II. However, it does not support as many features as ML-Rules, i.e., species cannot be attributed or nested and rate expressions are restricted to simple arithmetic calculations. SR networks also define continuous-time Markov chains (CTMCs), so that their trajectories can be computed with the stochastic simulation algorithm, see Section 2.4. Generally speaking, SR models refer more to the simple classical biochemical reaction networks [63] and therefore, many SSA variants have been implemented in JAMES II for SR: the Direct Reaction Method, the First Reaction Method, the Next Reaction Method, the Logarithmic Direct Method, and the Optimized Direct Method [97, 119]. For the Next Reaction Method [62], five event queues have been available for the experiments, `SimpleQueue`, `CalendarQueue`, `SimpleReBucketsQueue`, `MList`, and `PriorityQueue` (see [88]). Altogether, nine simulator configurations have been therefore available for the `Adaptive Simulator` to choose from.

To briefly summarize the experimental results, the `Adaptive Simulator` has also been able for SR to adapt the simulator configuration effectively and performed better compared to the random choice of the available simulator configurations. More details about the experiments done with SR are available in [75].

### 5.2.2 PDEVS

The parallel discrete event system specification (PDEVS) is a well-known modeling formalism for discrete event simulation that strictly separates between the model and the simulation algorithm called the abstract simulator [205, p. 75-77]. PDEVS is a hierarchical and modular modeling formalism, i.e., it provides formal means to compose atomic models to coupled models and to compose coupled models with

other atomic or coupled models. Eventually, the structure of a coupled PDEVS model can be described by a model tree, see Figure 5.17 (left). Based on a model tree, a processor tree can be generated that executes the simulation, see Figure 5.17 (right). The processor tree reflects the structure of the model tree and consists of coordinators (`C`) for each coupled model and simulators (`S`) for each atomic model. Coordinators forward messages of other coordinators and simulators. Simulators compute the behavior of atomic models. Only atomic models have explicit states. A root coordinator on top of the tree starts and ends the execution of a simulation step. The communication of coordinators and simulators is well-defined by a protocol and can be briefly explained as follows. Initially, the root coordinator sends a message that is forwarded to the simulator $S_i$ with the smallest next event time. The simulator $S_i$ is calculating an output forwarded to all simulators that are influenced by $S_i$. Afterward, $S_i$ and all influenced simulators are updating their states and they send the time of their next internal event to the root coordinator for the next simulation step.

For example, referring to the smoke detector model shown in Figure 5.17, a simulation step could be executed as follows. Suppose the current simulation time is 5, the battery wants to reduce its energy at time 10, and the speaker wants to give alarm at time 7. Firstly, the root coordinator sends a notification to the coordinator $C_{SmokeDetector}$ which is sending it to the simulator $S_{Speaker}$. After receiving the notification, the simulator $S_{Speaker}$ calculates an output which is sent to all influenced simulators through the coordinators. Suppose that an event of $S_{Speaker}$ influences the simulator $S_{Battery}$, because giving an alarm needs extra energy. Therefore, not only $S_{Speaker}$ is giving an alarm, but $S_{Battery}$ is also updating its state (reducing its energy level). Finally, both simulators send the time of the next internal event to the root coordinator and the simulation time is set to 7. For more details of this protocol and the execution of a simulation step, see [205].

Here, we focus on principle strategies to implement the processor tree [191]. For example, the *abstract threaded simulator* uses one thread for each coordinator and each simulator. On the one hand, this approach allows a full parallel execution. On the other hand, the number of threads on one machine is limited and the synchronization effort increases with an increasing number of threads. The thread limit problem could be solved by using a grid of computers, but this significantly increases the synchronization effort. The *abstract sequential simulator* avoids this problem by using one thread for the whole processor tree — the coordinators and simulators are processed sequentially [86]. However, the processor tree structure is still explicitly represented, i.e., one object per coordinator and one object per simulator. Another idea is to flatten the processor tree and consequently avoid individual objects for each coordinator and each simulator — the *flat sequential simulator* applies this concept.

Finally, the *parallel sequential simulator* partitions the model tree and applies a flat sequential simulator for each partition [85]. The flat sequential simulators can be executed in parallel. The number of partitions can be chosen to depend on the available hardware so that a suitable balance between parallelism and synchronization is achieved. Nevertheless, determining a suitable partitioning of the model tree is a complex task in itself and can decrease the benefits of the parallel execution. As usual, all these variants have their pros and cons: Which of these algorithms is the most suitable for a simulation run depends on the concrete model and on the available hardware and infrastructure.

For the experiment with PDEVS, we used a forest fire benchmark model [88]. This model represents a grid-based $100 \times 100$ forest and can be used do simulate how a fire spreads in this forest. Each grid area is defined as an atomic PDEVS model. The fire spreads until all areas of the forest are burned down. Each area passes trough three burning modes until it is burned down completely, i.e., four state transitions are executed for every area until it is burned down. All simulation runs were executed until all areas were burned down, i.e., $40,000$ simulation events have been executed per replication. We extended the original JAMES II model implementation described in [88] by adding stochasticity, i.e., the time that an area is inflamed by burning neighbors is stochastic.

We selected two simulators implemented in JAMES II for PDEVS for the `Adaptive Simulator`: an abstract threaded simulator and a flat sequential simulator [191]. Further, we selected five event queues that can be used by the simulators: `SimpleQueue`, `CalendarQueue`, `SimpleReBucketsQueue`, `MList`, and `PriorityQueue` [84]. Besides, the flat sequential simulator uses an event forwarding mechanism, for which two plugins have been available: `DirectExternalEventForwardingHandler` and `Hierarchical-ExternalEventForwardingHandler`. Thus, 15 simulator configurations have been available for the `Adaptive Simulator`. Moreover, only the number of burned down areas is considered for a base state $\sigma \in \Sigma$. For the state space representation, we used the static grid-based generalization with the grid size (1000), e.g., the aggregated state (1234) would be generalized to (1000) and the aggregated state (9876) would be generalized to (9000). Further, we again use $\epsilon$-decreasing with $\epsilon = 5$. We executed 100 inner replications with one inner thread and 50 outer replications with 10 outer threads.

Compared to the previous experiments with ML-Rules and SR, the initialization costs of both simulators are much higher compared to the initialization costs of the ML-Rules or SR simulators, i.e., the costs of an adaptation are much higher. Simulator objects for all atomic models have to be created and connections have to be established. Further, the event queue must be filled properly, i.e., all events of the simulator used

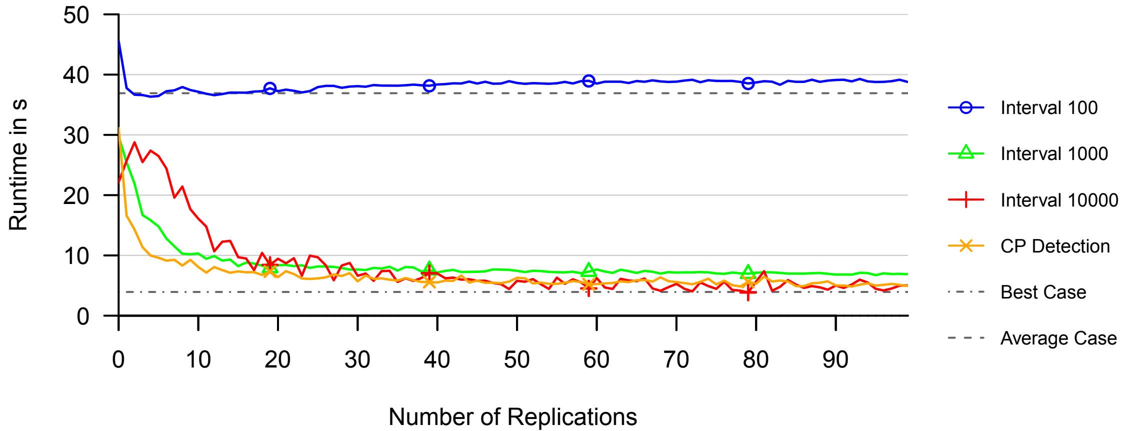



Figure 5.18: From [80]. Performance results of the forest fire model experiment. Due to high adaptation costs, too many adaptations are disadvantageous (blue line). Less adaptations can result in better performance values, but either a constant overhead remains (green line) or the learning rate slows down significantly (red line). The changepoint detection strategy (orange line) eliminates these issues and outperforms the other strategies.

before an adaptation have to be enqueued in the new queue.

The runtime results of the experiment are illustrated in Figure 5.18. We used three different static adaptation conditions, i.e., trigger an adaptation each 100, 1000, 10000 simulation events. In contrast to ML-Rules and SR, executing many adaptations with the forest fire model results in a significant overhead so that the `Adaptive Simulator` even performs worse than the average performance of all simulator configurations when executing an adaptation every 100 simulation events. The overhead is still noticeable with adaptations executed every 1000 simulation events ($\approx 2\ s$). Adapting every 10000 simulation events results again in a slower learning efficiency, but after 100 replications the performance is better compared to the higher adaptation frequencies.

Besides the static adaptation conditions, we also used the changepoint detection algorithm again with the four configurations constructed with $\delta \in \{10, 100\}$ and $h \in \{0.01, 0.0001\}$, see Section 5.1.2.3. Again, the results of the `Adaptive Simulator` with the four changepoint detection algorithm configurations have been similar, so that only one result is shown in Figure 5.18. By using the changepoint detection algorithm, the `Adaptive Simulator` performs better compared to the three static adaptation intervals. It learns faster and reduces the adaptation overhead in the long run because only few adaptations are executed ($\approx 10$ adaptations per replication). The number of changepoints evolves similar as in Figure 5.13. All in all, the experiment shows that

wrongly chosen static adaptation intervals can cause a disadvantageous performance and that our changepoint detection algorithm seems to be robust and effective.

## 5.3 Summary

In this chapter, the effectiveness and efficiency of the `Adaptive Simulator` has been analyzed by applying it to three modeling approaches: ML-Rules, SR, and PDEVS. Most experiments have been executed with the modeling language ML-Rules, which poses different computational challenges for the simulator, see Section 5.1.1. Based on these computational challenges, we developed a component-based ML-Rules simulator resulting in manifold configuration possibilities.

Using an ML-Rules benchmark model, we show that the `Adaptive Simulator` can effectively exchange its internal simulator during runtime eventually outperforming each simulator, see Section 5.1.2.1. Further, we explore the efficiency of different action selection policies and state space generalization methods with the benchmark model. Referring to the selection policies, $\epsilon$-decreasing performed best — it seems to be a simple but efficient policy, as the same observation has also been made in [50]. Referring to the state space generalization methods, fixed grids, the *Decision Boundary Partitioning Algorithm (DBPA)* and the *Adaptive Vector Quantization (AVQ)* have been applied. Fixed grids can perform well, but application dependent knowledge is necessary to configure the grid size suitably. The DBPA could outperform fixed grids, but it also sometimes failed completely resulting in a much worse performance. Finally, the AVQ has been more robust, but never performed as good as the DBPA.

Besides the ML-Rules benchmark model, we also used complex models applied in simulation studies to evaluate the `Adaptive Simulator` with ML-Rules: a Cell Cycle model [130], an Endocytosis model [74], and a Wnt/$\beta$-catenin pathway model [131, 132, 69]. Here, the `Adaptive Simulator` has been able to detect the best simulator at runtime, but it could not outperform it by executing adaptations. The models might not have different phases with different computational demands, so that one simulator is dominating the whole simulation run. Alternatively, although there might be different phases, one simulator can still dominate all the others resulting in the same observation. In both cases, the `Adaptive Simulator` cannot perform better than the best simulator, but it can detect this simulator and perform much better compared to the random choice of a simulator.

The Wnt/$\beta$-catenin pathway model has also been used to analyze the impact of different fixed adaptation intervals and the changepoint detection method, see Section 5.1.2.3. Whereas the changepoint detection method proved to be robust and efficient, the performance of the fixed intervals depends on the chosen interval

length. However, since an adaptation with ML-Rules is not computationally expensive, small intervals might be suitable to be used at all. In contrast, an adaptation of the PDEVS simulator is computationally expensive, and consequently, small intervals can significantly reduce the performance of the `Adaptive Simulator`, see Section 5.2.2.

Altogether, the `Adaptive Simulator` proved its ability to outperform simulators by executing adaptations during runtime. However, even in case one simulator dominates a simulation run, the `Adaptive Simulator` can detect this simulator and perform better than a random choice of a simulator. Eventually, the efficiency of the `Adaptive Simulator` depends on the available set of simulators. To extend this set for ML-Rules and consequently improve the potential of the `Adaptive Simulator`, we develop tailored and approximate simulators for ML-Rules and explore their efficiency in the next chapter.

# Chapter 6

# Tailored and Approximate Simulators - A Case Study with ML-Rules

> ...deducing general rules on the use of solvers, based only on the characteristics of the solvers themselves is not possible. The most appropriate solver also depends on the model (experiment) under consideration.
>
> Petra Claeys et al. [33]

Chapter 4 presents the concept of the `Adaptive Simulator` that uses an internal simulator to compute the actual model transitions and that adapts or exchanges this internal simulator as required. As it is integrated into the modeling and simulation framework JAMES II, it uses its registry to compute all possible simulators and configurations available to execute the current simulation run. Consequently, it can only exploit the options that are provided by the simulation system. As it cannot create new plugins on its own, the developers of plugins eventually have to implement tailored plugins for specific problems that are more efficient than more general plugins. Generally speaking, tailored algorithms are typically more efficient to solve a subset of problems of their problem domain $\mathbb{P}_{sub} \subset \mathbb{P}$, but they typically perform poorly or averagely for all other problems $\mathbb{P} \backslash \mathbb{P}_{sub}$. It might even be possible that a tailored algorithm is not able to solve all problems in $\mathbb{P}$, e.g., by aborting the calculation (throwing exceptions), or worse by calculating wrong results. In contrast, generic algorithms try to perform well for most problems of their problem domain $\mathbb{P}$. Referring to complexity, to put it in a nutshell, tailored algorithms have fantastic best-case complexities but disastrous average-case and worst-case complexities, whereas generic

algorithms have an average best-case and worst-case complexity.

Clearly, these thoughts can be applied in modeling and simulation for the development of tailored simulators, especially when models of a well-defined modeling language have to be executed. A tailored algorithm could require the model to be executed to fulfill specific properties, e.g., that it does not use all features provided by the language. In this case, it may be possible to avoid computationally complex calculations that are not needed, e.g., because some results are constant and have to be computed only once. Furthermore, the developer might be able to simplify the basic structure of the simulator. Nevertheless, such tailored simulators are not applicable for all models of the modeling language anymore. As written above, they might crash during the simulation by throwing exceptions or they produce wrong results.

Basically, there are two approaches to deal with this situation. First, one could analyze the model before executing it and check whether a given tailored simulator can be applied to execute this model correctly. To realize this idea, it is helpful to use a modeling language that allows comprehensive static analysis. For example, Petri Nets are not Turing complete and consequently allow complex analysis before executing them [157], e.g., it can be checked whether a transition is dead without any simulation. However, although Turing complete languages do not allow arbitrary complex analysis, it might still be possible to calculate important properties that are of interest for the simulators. Altogether, following the idea of tailored simulators requires the developers of the modeling languages and simulators to find ways to calculate properties of a concrete model before executing it. Further, for each simulator, it must be determined which properties are required to be applicable for this model. Fulfilling both requirements can be challenging, as a) some properties might be not computable, e.g., due to the Turing completeness of the used modeling language, and b) not all features required by a simulator might be identified by the developer, i.e., tailored simulators might be applied to models they are not applicable to and wrong results are calculated.

To circumvent these challenges, one could apply a more generic approach and let the simulation system automatically determine whether a simulator is valid for a given model. This can be done by executing pre-simulation runs, see Figure 6.1, e.g., as done for numerical integration solver by Claeys et al. [33]. The key idea is similar to the development of dynamic portfolios described by Gagliolo and Schmidhuber [59]. When a model shall be executed, all available simulators are run with this model in parallel or interleaved. Most simulators that are not applicable for the given model will quickly throw exceptions. Simulators that do not abort the calculation are executed until a specific condition is fulfilled, e.g., a specific simulation time is reached. The

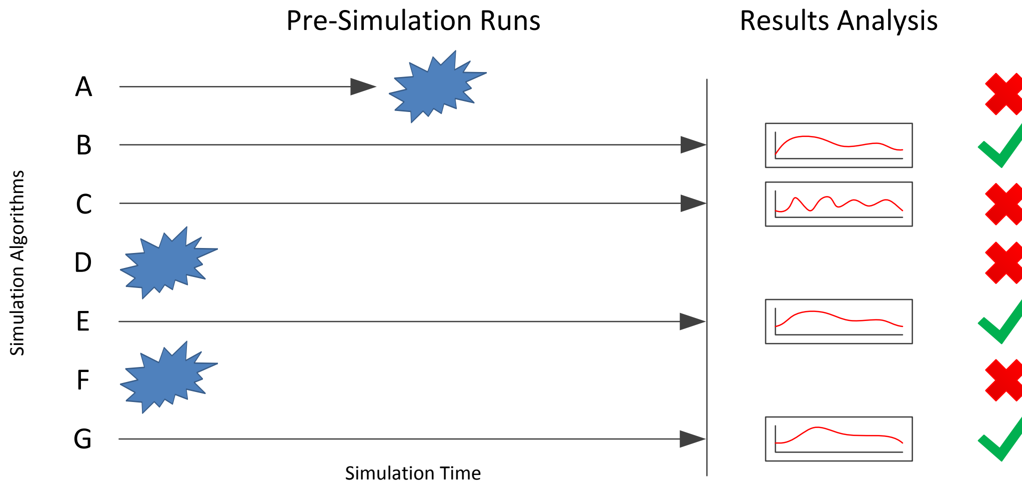



Figure 6.1: Illustration of an automatic validity check for simulators.

computed trajectories of all finished simulators are then compared and clustered. Since the correct trajectory is not known, a straightforward assumption is that the most frequent result is correct. All simulators producing this result are then concluded to produce correct results for the given model. Alternatively, one could require the results to fulfill specific conditions, e.g., described by temporal logics. This generic approach comes with its own particular challenges. The main problem is that *all* available simulators with all available plugin combinations and configurations have to be tested. It can simply be unfeasible to test all of them. Moreover, a simulator might produce correct results for the given pre-simulation run until a specific simulation time, but that does not guarantee that this simulator is valid for the given model. Consequently, applying this approach can lead to wrong simulation results. Further, in case of stochastic simulation runs, many replications would be needed to analyze the deviation of the produced results. Finally, it is difficult to compare results of approximate simulators, see Section 4.7.1.

In Section 5.1.1, the core features of ML-Rules and the induced computational challenges are presented. To tackle these challenges, we developed a component-based ML-Rules simulator and various components. The `Adaptive Simulator` has shown to be able to adapt the component-based ML-Rules simulator at runtime to improve the overall performance of simulation runs, see Section 5.1.2.1. For complex models used in simulation studies, we observed that one simulator dominates and therefore, the `Adaptive Simulator` cannot perform better than this simulator. To further improve the performance of ML-Rules simulation runs and the effectiveness of the `Adaptive Simulator`, in this chapter more ML-Rules simulators are developed and analyzed. First, in Section 6.1, we focus on simulators applicable to subsets of ML-Rules models,

which follow the semantics of the component-based ML-Rules simulator, i.e., producing the same results. Second, in Section 6.2 and Section 6.3, we focus on approximate simulators trading accuracy for speed.

## 6.1 Tailored Simulators for ML-Rules

The ML-Rules simulator must deal with various computational challenges, see Section 5.1.1. For example, the reaction set in ML-Rules is typically not fixed, i.e., reactions must be removed from and added to this set during runtime after each event execution, inducing a significant computational effort. However, not all ML-Rules models exploit the available features, e.g., models might have a fixed reaction set and therefore, the update operations can be avoided. In Section 6.1.1, we present a simulator for ML-Rules tailored to models with a fixed reaction set. Besides, bonds between entities play an important role for biochemical reaction networks. In Section 6.1.2, we present a tailored simulator for ML-Rules explicitly dealing with those bonds between entities.

Although achieving promising results, both tailored simulators are merely a first step towards the development of further tailored simulators inspired by simulators of other rule-based modeling languages for biochemical reaction networks. For example, NFSim is a network-free simulator developed for the modeling language BioNetGen [17] that avoids calculating the reaction network explicitly [179]. NFSim treats every species entity individually — the state of the system is a set of individuals — and links every individual to every reactant that it matches to. For each rule, the product of the link numbers for each reactant represents the number of potential reactions of this rule. For example, if 10 individuals match to the first reactant of a rule and 20 individuals match to the second reactant of the rule, $10 \cdot 20 = 200$ reactions are possible between these individuals. The reaction number of a rule is multiplied with the rate constant of this rule to calculate its propensity. Using the propensities of all rules, the SSA is applied for selecting a rule to be executed, see Algorithm 2.1 page 19. When selecting a rule, for each reactant one linked individual is chosen randomly to instantiate a concrete reaction to be fired. Finally, after firing a reaction, links are updated properly. All in all, network-free approaches perform particularly well if the number of rules is much smaller than the number of reactions. As these approaches avoid the calculation of the reaction set, they are interesting to be used for ML-Rules, as the maintenance of the reaction set causes most of the computational load. However, various challenges must be addressed, e.g., how to deal with reaction rates depending on reactant attributes.

Another example for an interesting simulator for ML-Rules is the simulator developed for the modeling language Kappa [37]. In Kappa, the *rigidity property* holds,

which implies that after matching one pattern of a connected pattern, the remaining matching process becomes clearly determined, i.e., only one valid mapping of patterns to species containing the specified matching exists [36]. Exploiting this property can speed-up the rule instantiation process, however, it does not hold for all ML-Rules models due to compartment connection of species entities representing hyperedges between entities [98].

### 6.1.1 Static Species and Reaction Sets

In contrast to the basic version of the stochastic simulation algorithm, the ML-Rules simulator has to deal with a changing reaction network. This part of the simulator induces most of the runtime of an ML-Rules simulation. However, we observed that some models developed in ML-Rules do not use all of its features and could be simulated more simply. For example, these models do not need the possibility to create, change, move and remove compartments during the simulation, so that the structure of model entities is fixed. Further, these models often do not use continuous attributes or complex attribute value calculations so that all possible attribute values for all species can be calculated before executing the simulation. Eventually, such models have a fixed reaction network and can be simulated more simply by calculating this reaction network *once* at the beginning of the simulation like it is done, e.g., in BioNetGen [17].

We developed a simulator for ML-Rules (`StaticSimulator`) that works in this manner and only supports models with a fixed reaction network [81]. This fixed reaction network has only to be calculated once at the beginning of a simulation run; only propensities of reactions must be updated during the simulation. Generally speaking, the simulator reduces to the basic SSA, see Algorithm 2.1 page 19. To guarantee a fixed reaction set, expressions and function calls are not allowed to be used neither within the reactants nor within the products of rules. Further, compartments are neither allowed to be created, nor to be removed, nor to be changed. Finally, the simulator can only be applied when compartments are treated individually, i.e., a population-based treatment of compartments is not supported. Otherwise, due to splitting and merging processes, the reaction set would have to be updated frequently, i.e., the reaction set would not be fixed. Altogether, these requirements guarantee a reaction network to be closed under reaction execution. In the following, we refer to reactions fulfilling these requirements as *static* reactions.

Although compartments are not allowed to be created or removed, *structure-preserving* multi-level rule schemes are supported by the `StaticSimulator`. For example, the rule scheme

```
Nucleus[s?] + BetaCatenin -> Nucleus[BetaCatenin + s?] @ ...;
```

describes the shuttling of a (non-compartmental) `BetaCatenin` into a `Nucleus` without changing the structure of the model state. However, when executing a reaction based on this rule scheme, the standard simulator of ML-Rules would remove an existing `Nucleus` (and its content $s?$) and one `BetaCatenin` from their context and it would create a new `Nucleus` including the solution $s?$ and one `BetaCatenin`. Therefore, the reaction network is updated afterwards, i.e., all reactions using the old `Nucleus` or its content have to be removed and new reactions for the created `Nucleus` and its content must be determined.

The `StaticSimulator` does not follow this behavior: it considers only non-compartmental entities when executing such structure-preserving reactions. Consequently, when executing a reaction based on the shuttling rule scheme, the `StaticSimulator` would simply decrease the amount of `BetaCatenin` in the context of the `Nucleus` and it would increase the amount of `BetaCatenin` in the `Nucleus`. Thus, the `Nucleus` is not changed directly and only propensities of reactions must be updated; the reaction network itself does not change. Such associations between reactant and product compartments, i.e., to identify that the reactant `Nucleus` *is* the product `Nucleus`, can be determined automatically via a static model analysis before a simulation run.

#### 6.1.1.1 Results with the Wnt/$\beta$-catenin Pathway Model

Figure 6.2 (top) shows the runtime results for simulations of a Wnt/$\beta$-catenin pathway model [132] implemented in ML-Rules with different number of simulated cells until simulation time 300[1]. Figure 6.2 (bottom) shows averaged simulation results of the simulators when simulating eight cells. To simulate one cell, the `StandardSimulator` without a dependency graph performs worst (1 cell $\approx$ 43 s, 10 cells $\approx$ 2870 s). The runtime increases polynomially with an increasing number of cells to be simulated. This polynomial runtime growth is reduced to an almost linear growth by using the dependency graph (see Section 5.1.1.2 page 102), because the cells of the model do not interact directly with each other, but only via the Wnt protein. The runtime behavior of the simplified simulator (`StaticSimulator`) is similar, i.e., it is polynomial without a dependency graph and linear with a dependency graph. Consequently, the `StaticSimulator` without a dependency graph will eventually perform worse with an increasing number of cells than the `StandardSimulator` with a dependency graph. This result shows that it is still worth to research and develop general optimizations applicable to all ML-Rules models. Finally, the `StaticSimulator` with a dependency

[1]Experiment computer: *Intel(R) Core(TM) i7 CPU X990 @ 3.46 Ghz* with activated Hyper-threading and deactivated TurboBoost, 24GB RAM, Windows 7 and Java 8

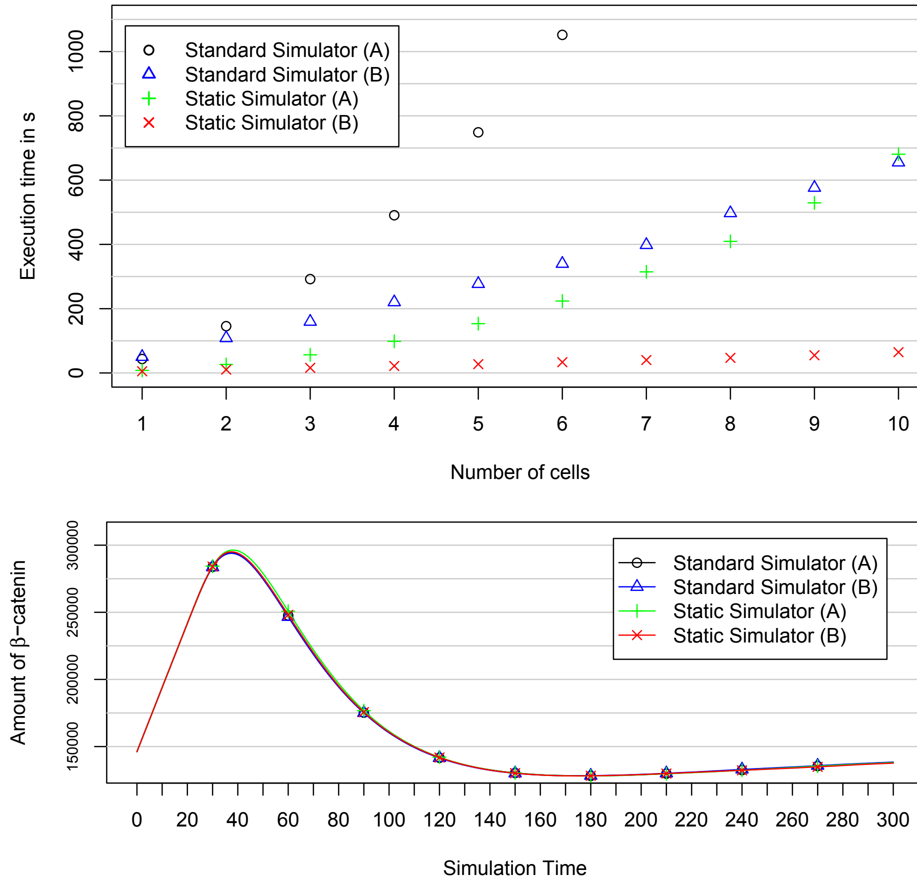



Figure 6.2: From [79]. Top: Average runtime of the different ML-Rules simulators for the Wnt/$\beta$-catenin pathway model until simulation time 300 based on 20 replications. Simulators without a dependency graph are denoted by $A$, simulators employing a dependency graph by $B$. Bottom: Simulation results when simulating eight cells.

graph performs best (1 cell ≈ 5 s, 10 cells ≈ 65 s), i.e., for 10 cells, it is more than 40 times faster than the `StandardSimulator` without a dependency graph and still 10 times faster than the `StandardSimulator` with a dependency graph.

### 6.1.2 Species Bindings

Besides the `StaticSimulator` that is tailored to ML-Rules models with a static reaction set, we developed a `LinkSimulator` that is tailored to ML-Rules models focusing on bindings between species. Bindings often play an essential role in many biochemical systems like mitochondrial networks [203]. In ML-Rules, bindings have to be represented by attribute values, i.e., no explicit binding construct exists in ML-Rules. The following rule schemes illustrate how species of type `MitoA` and `MitoB`

```
1  // species definitions
2  MitoA(link)[];
3  MitoB(link)[];
4
5  // initial solution
6  >>INIT[100 MitoA(free) + 100 MitoB(free)];
7
8  // rules
9  MitoA(free) + MitoB(free) -> MitoA(x) + MitoB(x)
10     @ 1 where x = nu();
11 MitoA(x) + MitoB(x) -> MitoA(free) + MitoB(free) @ 1;
```

Figure 6.3: A simple model in which species can bind and unbind.

could be bound and unbound:

```
(1) MitoA(free) + MitoB(free) -> MitoA(x) + MitoB(x) @ ... where x = nu();

(2) MitoA(x) + MitoB(x) -> MitoA(free) + MitoB(free) @ ...;
```

In the first rule pattern, the function `nu()` returns a unique value so that after a reaction firing the two selected `MitoA` and `MitoB` entities share a unique attribute value that represents their binding. For these unique binding values, we added the attribute type `link` to the set of available attribute types. Thus, only bound entities match both reactants of the second rule pattern and can therefore be unbound. Especially the second rule pattern is interesting because after selecting a concrete `MitoA` entity for the first reactant, the second entity is clearly determined, i.e., the *rigidity property* holds see Section 6.1 page 131. Therefore, the simulator can directly determine all reactants after matching one concrete reactant of this rule. The `LinkSimulator` exploits this property by saving all bound pairs of entities explicitly in an additional data structure to directly access bound partner. Nevertheless, this procedure can be improved by applying a further technique we refer to as *Reactant Swapping*. Besides, the simulator does not support to change bonds in functions on solutions, as the additional data structure would not be maintained. Further, a unique bond does only connect exactly two entities — in principle a unique bond can connect more entities in ML-Rules.

#### 6.1.2.1 Reactant Swapping

*Reactant Swapping* is a technique used by the `LinkSimulator` to improve the reaction creation process. We illustrate this technique with a simple mitochondria model, see Figure 6.3. The model uses the compartment species types `MitoA` and `MitoB` (both

with one `link` attribute) and two rule schemes to bind and unbind entities. Since the species types refer to compartments, all entities are treated individually. Further, suppose that an instantiation of the binding rule scheme (ll.9-10) has been fired lastly, i.e., one `MitoA` and one `MitoB` are bound. When updating the reaction set **R** after firing this reaction, firstly invalid reactions are removed from the reaction set. All reactions containing a species entity changed by the fired reaction are invalid. Referring to the example, all reactions of the binding rule scheme that uses one of the meanwhile bound `MitoA` or `MitoB` have to be removed. Next, new reactions are calculated. Every new reaction must at least contain one species entity that has either been modified or been created by the fired reaction (in the following the term *changed* is used to refer to modified or created entities). Otherwise, old reactions would be recalculated. For the first rule scheme, no new reactions will be found, since no unbound entity has been changed (and is still unbound) or created. However, new reactions can be created based on the second rule scheme (l.11). In general, the reaction creation for a rule scheme processes successively from the leftmost reactant to the rightmost reactant. Thus, all entities that match the first reactant are initially identified, i.e., all bound `MitoA` entities match this reactant. For the second reactant, two cases must be distinguished. First, if the changed `MitoA` is selected for the first reactant, a valid entity for the second reactant only has to match this reactant, no matter whether it is changed or not. Since `link` values are unique, only one matching `MitoB` is found and one reaction can be instantiated. Second, if one of the 99 available non-changed `MitoA` entity is selected for the first reactant, the `MitoB` entity matched to the second reactant must be changed; otherwise only an already calculated reaction would be calculated again. For this case, no reactions are found, since the only changed `MitoB` only fits to the changed `MitoA`.

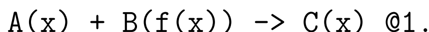
This approach is not optimal since all the 99 checks of the second case are not necessary. By applying *Reactant Swapping* the simulator avoids this case. Algorithm 6.1 illustrates the approach of *Reactant Swapping*. The basic idea is to consider only changed entities for the first reactant of the current reactants list (l.4). In this case, the condition that at least one matched entity is changed for new reactions is already fulfilled. Next, the remaining reactants are checked using the found matched entities for the first reactant (l.6). The first reactant is then moved to the last position of the reactants list (l.8). To avoid calculating the same reactions several times, e.g., if a reaction contains two changed entities, the function `findAllReactions()` uses the counter `i` as a parameter and restricts the last `i` reactants to be *non-changed*. Referring to the example and the second rule scheme, initially the changed `MitoA` would be selected for the first reactant and the changed `MitoB` would be directly selected for the second reactant (due to the additional data structure maintaining

**Algorithm 6.1** Sketch of the *reactant swapping* principle written in Java.

```java
1  List<Reaction> reactions = new ArrayList<>();
2  for (int i = 0; i < reactants.size(); ++i) {
3    // find all matching changed species for the first reactant
4    List<Matching> m = findMatchings(changedSpecies, reactant.get(0));
5    // calculate all reactions based on the found matched species
6    reactions.addAll(findAllReactions(m, i, reactants));
7    // remove the first reactant from the reactant list
8    Reactant tmp = reactants.remove(0);
9    // add the removed reactant to the end of the list
10   reactants.add(tmp);
11 }
```

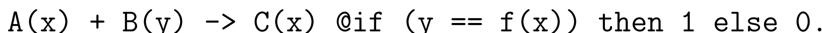
all bound pairs used by the `LinkSimulator`). Next, the `MitoA(x)` reactant would be moved to the end and the matching would start again with the new first reactant, i.e., `MitoB(x)`. Here, only the changed `MitoB` is found as a changed matching entity and for the second reactant, only the changed `MitoA` is found. Therefore, no additional reaction would be created, since no non-changed matching `MitoA` can be found. All in all, due to *Reactant Swapping*, the number of matching processes is decreased from 100 to 2 checks.

However, *Reactant Swapping* does not dominate the default version in all cases. For example, ML-Rules allows using expressions to restrict attribute values of matched species entities for reactants, e.g.,



```
A(x) + B(f(x)) -> C(x) @1.
```

*Reactant Swapping* cannot be applied directly in this case, since the second reactant `B(f(x))` cannot be considered before matching the reactant `A(x)`. However, every rule scheme of this form can be rewritten to avoid this issue, i.e., the restriction of the attribute value can be moved to the rate expression:



```
A(x) + B(y) -> C(x) @if (y == f(x)) then 1 else 0.
```

Nevertheless, the disadvantage of this rewritten rule scheme is that the time to reject an invalid reactant combination is now postponed to the rate calculation. Further, although *reactant swapping* can also be applied for the standard simulator of ML-Rules, model rewriting would also be necessary.

#### 6.1.2.2 Results with a Mitochondria Model

Figure 6.4 shows runtime results for simulations of a simple mitochondria model [16] implemented in ML-Rules with different number of mitochondria until simulation

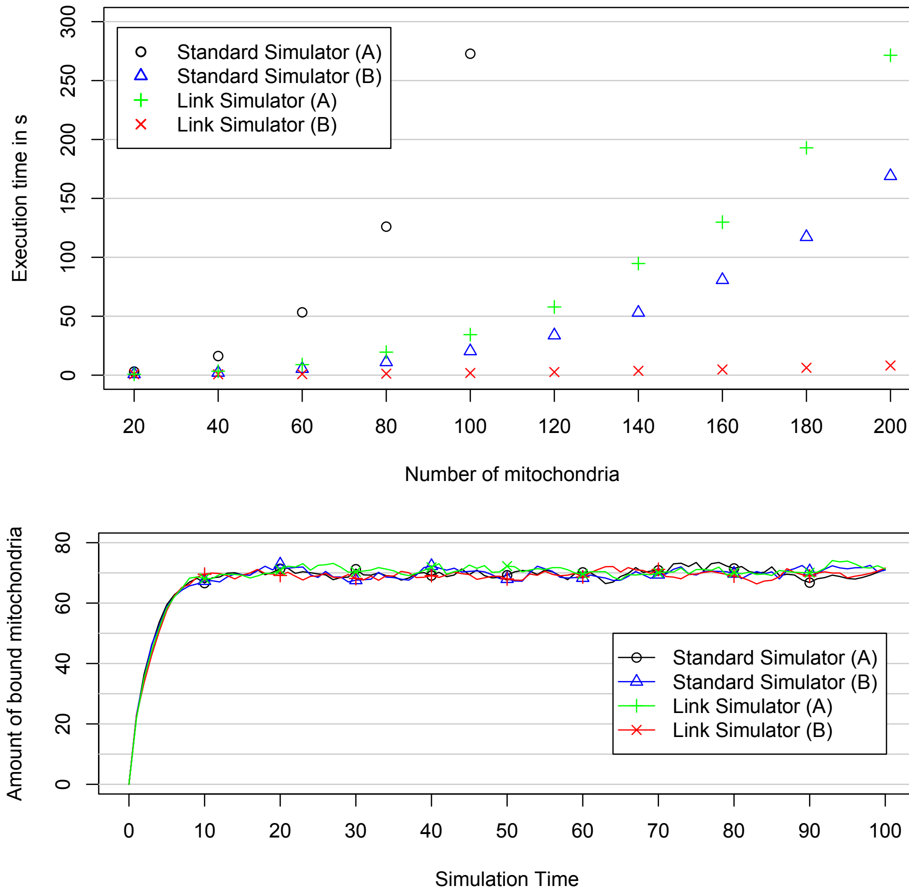



Figure 6.4: From [79]. Top: Average runtime of the standard ML-Rules simulator and the tailored simulator for the ML-Rules subclass $M_{bind}$ simulating the mitochondria model until simulation time 100 based on 20 replications. Simulators without a dependency graph are denoted by $A$, simulators employing a dependency graph by $B$. Bottom: Simulation results of the simulators when simulating 200 mitochondria.

time 100[2]. In this model, bindings of mitochondria are represented as explained above by unique attribute values of type `link`. Several rule schemes change two bound mitochondria, i.e., they fulfill the rigidity property. Again, results of the standard simulator with and without a dependency graph are shown. The `LinkSimulator` has also been used with and without a dependency graph. Analog to the results of the Wnt/$\beta$-catenin pathway model, the dependency graph is a useful optimization that significantly reduces the runtime of the simulation runs. Further, the tailored `LinkSimulator` is significantly more efficient than the standard simulator.

[2]Experiment computer: *Intel(R) Core(TM) i7 CPU X990 @ 3.46 Ghz* with activated Hyper-threading and deactivated TurboBoost, 24GB RAM, Windows 7 and Java 8

## 6.2 $\tau$-leaping for ML-Rules

The `StaticSimulator` and the `LinkSimulator` are still *exact* in the sense of the SSA (for the models the specialized simulators are applicable to). Besides implementation improvements of the ML-Rules simulator not changing simulation results, another approach is to use approximate simulators trading accuracy for efficiency. A well-known approximate algorithm is $\tau$-leaping [64], see Section 2.4 page 20ff. This algorithm performs "leaps" along the time line and approximates the number of reaction firings during these leaps. The basic assumption is that the propensities of all reactions do not change significantly during a leap. This allows approximating the number of reaction firings during the leap with a Poisson distribution. The length of a leap is limited by the *leaping condition* (see Definition 8, page 21) bounding the induced error by approximating the change of the reaction propensities.

In [78], we present a $\tau$-leaping simulator for ML-Rules supporting population-based compartments based on the $\tau$-leaping variant presented by Cao et al. in [25], which works as follows. Initially, the current reaction set $\mathbf{R}$ is determined. Afterward, the reactions are separated to critical reactions $\mathbf{R}_{cr}$ and non critical reactions $\mathbf{R}_{ncr}$. Referring to ML-Rules, a reaction is a non critical reaction if it can be fired more often than $n_c \in \mathbb{N}$ times and if it does not change the reaction set, i.e., it fulfills the requirements for reactions defined for the `StaticSimulator`, see Section 6.1.1. Further, *structure-preserving* rule schemes are treated in the same way as done by the `StaticSimulator`, i.e., compartments are not removed and added when executing reactions based on these rule schemes.

After the reaction separation, the $\tau'$ candidate for the non-critical reactions is calculated. Hierarchical multiplicities of the propensity calculation have to be ignored, since the calculated mean and variance (see Equation 2.4) refer to the change of a reactant species in an individual context. This change is therefore independent from the copy number of the contexts up to the root. If $\tau'$ is too small, i.e., smaller than $\alpha \cdot a_0(X(t_j))$, no leap but $N_{SSA} \in \mathbb{N}$ SSA steps are executed. In case $\tau'$ is sufficiently large, a second candidate $\tau''$ is calculated using the set of critical reactions, see Equation 2.5. Finally, $\tau = min(\tau', \tau'')$ and if $\tau'' < \tau'$, one critical reaction is selected to be executed during the next $\tau$-leap.

Firing numbers for reactions are sampled from a Poisson distribution, see Equation 2.1. For each reaction $R_i$, the rate of the used Poisson distribution is $a_i(X(t_j)) \cdot \tau$. However, when dealing with population-based compartments, one sampled firing number would be applied to *all* individual compartments represented by the context entity of the reaction, see Figure 6.5 (**A**). This can reduce the accuracy of the simulation results. On the other hand, treating each compartment individually would make the population-based approach more or less useless, see Figure 6.5 (**B**). To set the

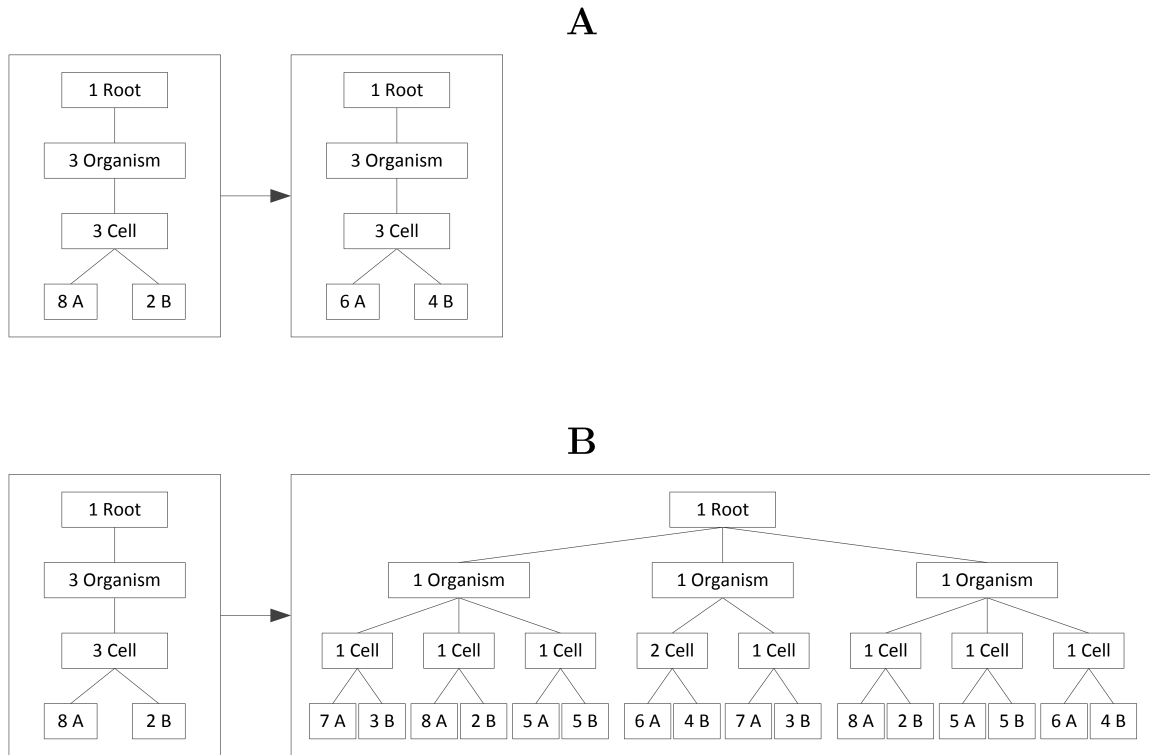



Figure 6.5: **A**: Suppose the reaction `A -> B` is fired in the solution on the left during a $\tau$-leap. All identical compartments are treated equally, i.e., only one firing number for the reaction `A -> B` is sampled (in the shown case the sampled firing number is 2) and applied. **B**: Again the reaction `A -> B` is fired in the solution on the left during a $\tau$-leap. Every compartment is treated individually, i.e., for each compartment a firing number for the reaction `A -> B` is sampled and applied. Note that since firing numbers can be identical, a merging process is also applied to merge identical result compartments.

degree of "individuality" for compartments, we introduce a parameter $\mu \in \mathbb{N} \cup \infty$. A compartment $c$ with copy number $|c|$ is divided into $\mu$ groups of size $\frac{|c|}{\mu}$ treated individually. Therefore, if $\mu = 1$, all compartments of the current solution evolve equally, like shown in Figure 6.5 (**A**). In contrast, if $\mu$ is greater than the copy number (guaranteed by $\mu = \infty$), all compartments are treated individually. When using $\mu$, the following procedure to calculate a $\tau$-leap must be processed top-down beginning from the root context:

1. For each reaction $R_i$ in the current context, sample a firing number $fn_i$ from a Poisson distribution with rate $a_i(X(t_j)) \cdot \tau$ and execute $R_i$ $fn_i$ times.

2. For each compartment $c$ in the current context, create $min(|c|, \mu)$ groups.

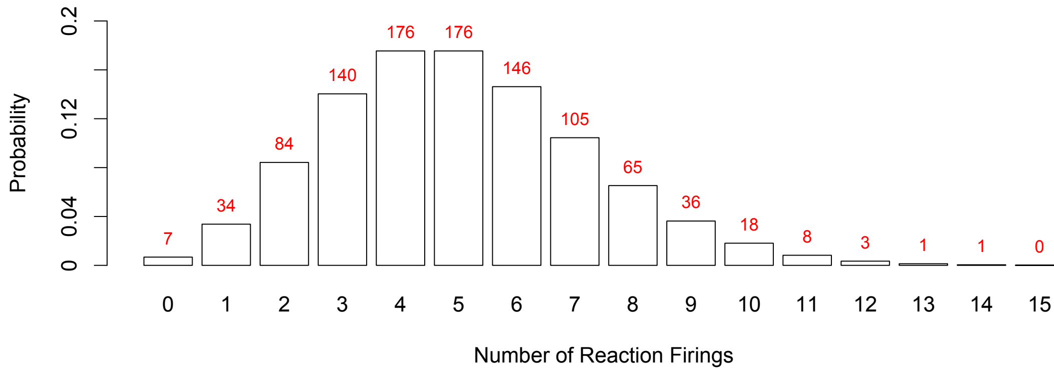



Figure 6.6: Probabilities of firing numbers for one reaction $R_i$ : `A -> B` with rate $a_i = 2$ and $\tau = 2.5$ and the corresponding number of `Cell` compartments (red numbers) assigned to the firing numbers.

3. For each group of each compartment, repeat this procedure.

Afterward, a merging process has to be processed bottom-up in each compartment to merge identical compartments resulting in a valid population-based solution.

Alternatively to $\mu$, one could also directly apply the probabilities of each firing number to calculate the "individuality" of compartments. For example, Figure 6.6 shows the probability distribution of firing numbers for the solution `1000 Cell[10 A + 10 B]`, the reaction $R_i$ : `A -> B` with rate $a_i = 2$, and $\tau = 2.5$. The number of `Cell` compartments for each firing number is shown on top of each bar, i.e., 15 groups of `Cell` compartments would be created. In case more than one reaction is possible in a compartment, assuming that the reaction numbers of each reaction are independent, independent Poisson distributions have to be combined to calculate the probabilities of the firing number tuples.

Figure 6.7 illustrates the number of `Cell` compartments for each firing number tuple when adding a second reaction $R_k$ : `B -> A` with rate $a_k = 2$ to the example. For instance, the tuple $(4, 5)$ (reaction $R_i$ is fired 4 times and reaction $R_j$ is fired 5 times) would be applied to 31 `Cell` compartments. Altogether, following this approach quickly results in an individual-based treatment of compartments with in increasing number of reactions. This is already indicated in the example by the numbers of distinguished groups of `Cell` compartments with one reaction (resulting in 15 groups) and two reactions (resulting in 131 groups). However, when using the probabilities of the Poisson distribution to group the compartments, the simulator does not necessarily

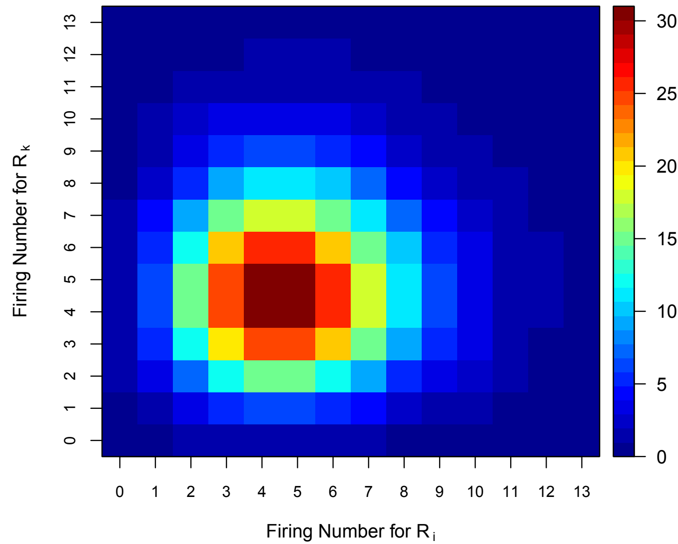



Figure 6.7: By combining two independent Poisson distributions for the reactions $R_i$ and $R_k$, the number of `Cell` compartments (identified by the color key) for each firing number tuple can be calculated. For example, `31 Cell` compartments would have the firing number tuple $(4, 5)$.

have to process a $\tau$-leap top-down. It is probably even more efficient to process bottom-up, because the change of a compartment is identical for all individuals of its context and therefore should only be computed once.

Note that independent from the concrete mechanism, when a critical reaction shall be executed, an individual branch of its context must be extracted to ensure that the reaction is only executed once. Finally, if the model state is invalid after a $\tau$-leap, i.e., the amount of at least one species is negative, the changes are discarded, $\tau'$ is halved, and the algorithm is repeated.

In general, the error approximation for the calculation of $\tau'$ and $\tau''$ bases on the assumption that all reactions refer to mass action kinetics, i.e., the propensity of a reaction is the result of a multiplication of the reactant amounts and the rule constant. In ML-Rules, however, arbitrary rate equations can be formulated, so that small changes of species amounts might result in significant and unpredictable changes of propensities. For example, by using `if` ... `then` ... `else` ... blocks, the rate equation might return significant different values depending on whether the condition of the such a block is `true` or `false`. Therefore, when using complex rate equations, the error estimation of $\tau$-leaping for ML-Rules can fail — a problem that makes it impossible for the `Adaptive Simulator` to rely on the error approximation.

Analog to the SSA execution of ML-Rules models, compartments typically diverge

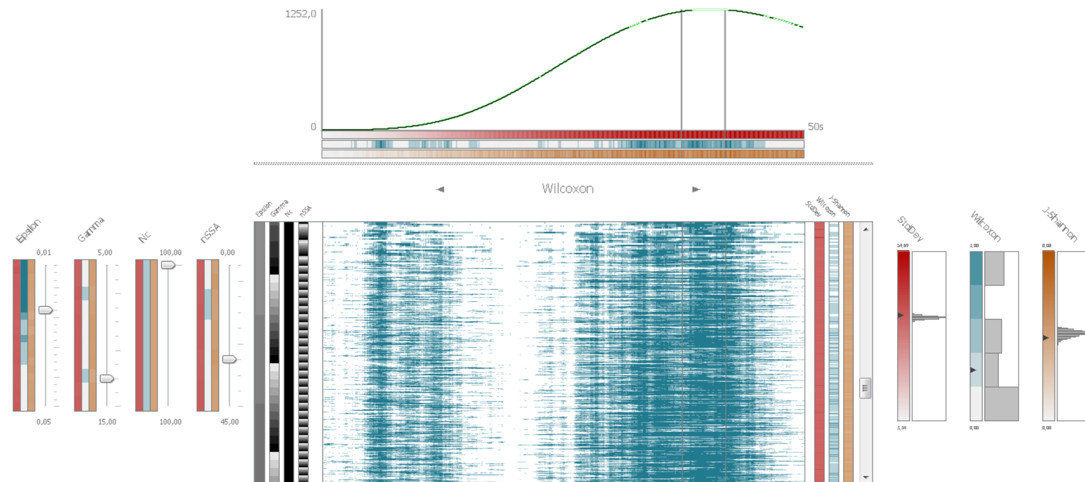



Figure 6.8: Screenshot of the visual analytics tool for accuracy analysis presented in [123]. Top: Concrete simulation results of a $\tau$-leaping configuration (average of $n$ replications) compared to the reference results computed by the SSA. Left: Bars to select a concrete $\tau$-leaping configuration. Right: Histograms for different accuracy measurements (the standard deviation, the p-value of the paired Wilcoxon rank sum test [176, p. 513], and the Jensen-Shannon distance [121]) for all $\tau$-leaping configurations. Middle: results of one selected measurement for each time point and a set of $\tau$-leaping configuration (one line represents one $\tau$-leaping configuration). The order of the configurations can be changed (by accuracy values or parameter values).

during a simulation and therefore, the $\tau$-leaping simulator will probably quickly treat all compartments individually (whether $\mu$ is used or not). Therefore, we also developed a simplified $\tau$-leaping variant for ML-Rules only supporting individual-based compartments. This variant does not need a grouping mechanism for compartments with a parameter like $\mu$ or the multivariate Poisson distribution. Further, no splitting and merging procedures are necessary.

### 6.2.1 Results and Accuracy Analysis with Visual Analytics

To evaluate $\tau$-leaping for ML-Rules, we used the Wnt/$\beta$-catenin pathway model [131, 132, 69] and the cell cycle model [130] we already used for the `Adaptive Simulator` (see Appendix A.3 and A.1) and a simplified version of a lipid raft model describing the synthesis, degradation and diffusion of lipid rafts in cell membranes [68], see Appendix A.4 for the ML-Rules implementation of this model.

We used 480 configurations to analyze the performance of $\tau$-leaping built from the cross product of the following parameter values:

- $\epsilon \in \{0.01, 0.02, \ldots, 0.2\}$ (error acceptance parameter)

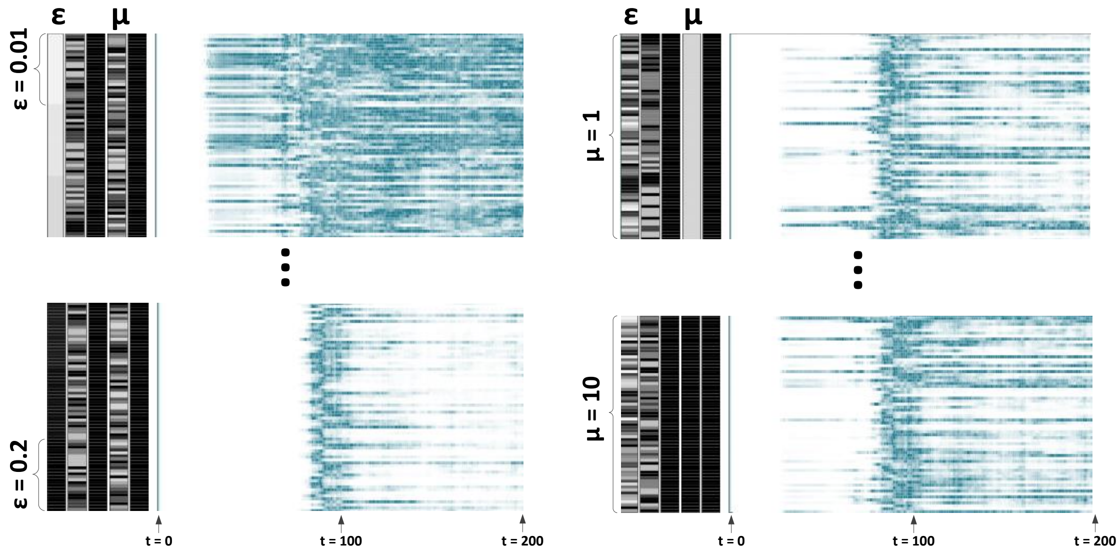



Figure 6.9: From [78]. Accuracy illustrations of parameter settings sorted by the value of the parameter $\epsilon$ (left) and by the value of the parameter $\mu$ (right) on the example of the dephosphorylized Axin proteins of the Wnt/$\beta$-catenin pathway model with ten cells. Accuracy values represented by p-values of the paired Wilcoxon rank sum test [176, p.513] are mapped from white (low values) to saturated cyan (high values). Whereas $\epsilon$ clearly influence the accuracy (left), $\mu$ seems not to have an impact (right).

- $\alpha \in \{5, 10, 15, 20\}$ (threshold factor for $\tau$ to be accepted)
- $\mu \in \{1, 4, 6, 8, 10\}$ (group parameter for population-based compartments)
- $n_c = 10$ (minimum firing number of reaction to be non critical)
- $N_{SSA} = 100.$ (number of SSA steps in case of small $\tau$ values)

For each configuration and each model, we executed 100 replications for both the analysis of the simulation trajectories and the runtime performance[3]. Each replication has been executed until the simulation time 200.

The best results have been achieved with the Wnt/$\beta$-catenin pathway model. Due to the fixed structure of this model (the compartments of the model are not changed) and mainly shuttling events, $\tau$-leaping can execute large leaps and critical reactions are rare. When simulating one cell, the ML-Rules SSA needed $\approx 39s$ on average, the fastest $\tau$-leaping configuration ($\epsilon = 0.2$, $\alpha = 5$, $\mu = 1$) only needed $\approx 0.5s$ on average, and the slowest $\tau$-leaping configuration ($\epsilon = 0.01$, $\alpha = 5$, $\mu = 10$) needed

[3]Experiment computer: *Intel(R) Core(TM) i7 CPU X990 @ 3.46 Ghz* with activated Hyper-threading and deactivated TurboBoost, 24GB RAM, Windows 7 and Java 7

$\approx 1.8s$ on average, i.e., even with the slowest $\tau$-leaping configuration, the runtime has been reduced by more than 95%. When simulating ten cells, the ML-Rules SSA needed $\approx 1125s$ (ca. thirty times more runtime compared to the simulation of one cell), however, $\tau$-leaping still needs less than $15s$ for all configurations. Therefore, in this case $\tau$-leaping even reduced the runtime by more than 98%. Since the cells in this model do not interact directly with each other, but only indirectly with the `Wnt` species, the leap sizes are similar compared to the one cell case, whereby the SSA has to execute ten times more reactions. All parameters but $\epsilon$ had little impact on the performance. The parameter $\mu$ almost had no impact since all cells diverse quickly, i.e., the model quickly evolves to states with individual-based compartments.

For the analysis of the accuracy of the simulation results, we used a visual analytics tool [123], see Figure 6.8. This tool enabled us to visually validate the accuracy of simulation results of the $\tau$-leaping configurations with respect to simulation results computed by the SSA. As expected, the error parameter $\epsilon$ mostly influenced the accuracy of the simulation results, whereas the parameter $\mu$ had little impact, see Figure 6.9. The vertical white lines in the diagrams are caused by inaccurate reference results, i.e., the reference values are also only approximations calculated by 100 SSA replications.

Although the results with the Wnt/$\beta$-catenin pathway model are promising, $\tau$-leaping did not achieve similar results neither with the lipid raft model nor with the cell cycle model. For the lipid raft model, the accuracy of the results was poor for most configurations, only configurations with a small $\epsilon$ achieved a suitable accuracy. These configurations could reduce the runtime by $\approx 15\%$ (the SSA needed $\approx 72s$). Referring to the cell cycle model, $\tau$-leaping did not even achieve any improvement. Most reactions of this model refer to changes of cells, which are represented by compartments. Therefore, most reactions are critical reactions and $\tau$-leaping mostly executed only one critical reaction and no non-critical reactions during a leap.

## 6.3 Hybrid Simulator for ML-Rules

Besides $\tau$-leaping, hybrid simulators combining deterministic and stochastic methods are a common approach to approximately simulate biochemical reaction networks, see Section 2.4.2. Therefore, we also developed a hybrid simulator for ML-Rules. Two components of a hybrid simulator for biochemical reaction networks have to be distinguished: 1) A component partitioning the reactions and 2) a component approximating the deterministic reactions.

**Algorithm 6.2** Calculate the threshold $\alpha$ for the separation of slow and fast reactions. The reaction and propensity indices are assumed to be sorted by propensities, i.e., $a_1(X(t_j))$ is the lowest propensity and $a_{|R|}(X(t_j))$ is the highest propensity.
$K \in [0, 1]$: minimum relative distance of fastest stochastic reaction and slowest deterministic reaction,
`isStatic(`$R_k$`)`: returns true, if $R_k$ is static; otherwise false,
$t_j$: simulation time of the $j$-th step.

```
1 for ((a_i(X(t_j)), a_{i+1}(X(t_j))) ∈ ((a_1(X(t_j)), a_2(X(t_j))), ..., (a_{|R-1|}(X(t_j)), a_{|R|}(X(t_j))))) {
2   if (a_i(X(t_j))/a_{i+1}(X(t_j))) > K) {
3     if ({R_k|R_k ∈ (R_i, ..., R_{|R|}) ∧ !isStatic(R_k)} = ∅) {
4       return a_i(X(t_j));
5     }
6   }
7 }
8 return ∞;
```

## 6.3.1 Reaction Partitioning

Inspired by [35], we apply the requirements defined for reactions of the `StaticSimulator` (see Section 6.1.1) to partition reactions for the hybrid simulator of ML-Rules. Concretely, all static reactions are calculated deterministically; all other reactions are treated stochastically. Using this property to determine deterministic reactions is useful since the set of deterministic reactions is in this case closed under reaction execution. The set of deterministic reactions is referred to as $R^d = \{R^d_1, \ldots, R^d_{|R^d|}\}$. The set of stochastic reactions is referred to as $R^s = \{R^s_1, \ldots, R^s_{|R^s|}\}$. Accordingly, the propensity functions are named $a^d_1$, $a^s_1$, etc.

In addition to this separation strategy, as commonly done by many hybrid simulators, we also integrated an adaptive scheme to consider propensities of reactions to partition them. First, a threshold $\alpha \in \mathbb{R}$ is calculated based on all propensities, see Algorithm 6.2. Basically, $\alpha$ is the lower element of the first pair of consecutive propensities $(a_i(X(t_j)), a_{i+1}(X(t_j)))$ with a relative distance that is larger than a user-defined parameter $K$. Since all reactions with a propensity greater than $\alpha$ shall be deterministic, they must also be static. The presented approach does not explicitly consider models with more than two time scales, but only separates the slowest group of reactions from the others. If $\alpha = \infty$, no clear separation between fast and slow reactions is possible. In this case, the group of fast reactions is empty and the simulator degrades to the SSA. Altogether, considering the propensities makes the partitioning more restrictive, but it should be less error-prone since it avoids fast reactions that are calculated stochastically and slow reactions that are calculated deterministically. Nevertheless, it induces a regular overhead that might not be necessary.

### 6.3.2 Calculation of Deterministic Reactions

For the approximation of deterministic reactions, we developed two methods. The first method is inspired by the hybrid simulator presented by E et al. in [40], see Section 2.4.2. This method assumes that for every species involved in deterministic reactions, a stationary distribution exists when only considering deterministic reactions. In the following, we refer to these species changed by deterministic reactions as deterministic species. Further, these stationary distributions must be reached much faster compared to the next firing time of a stochastic reaction. The hybrid simulator presented by E et al. in [40] uses pre-simulation runs considering the deterministic reactions to approximate the stationary distributions. We avoid such pre-simulation runs by applying the following approach. All deterministic reactions are executed until steady states are observed for all deterministic species, see Figure 6.10 (left). We use a crossing mean steady state estimator to decide whether a steady state has been reached [199]. If at least for one deterministic species, no steady state is detected after $n_1 \in \mathbb{N}$ simulation steps, the hybrid simulation step is aborted and the simulation proceeds instead with $n_2 \in \mathbb{N}$ SSA steps. Otherwise, $m \in \mathbb{N}$ SSA steps are executed only considering the deterministic reactions. The distribution of each deterministic species is recorded and used to approximate the stationary distributions, see Figure 6.10 (right). Clearly, the larger $m$ is selected, the more accurate get the approximations. After approximating the stationary distributions, the propensities of the stochastic reactions are updated accordingly, see [25] (Equation 9). Next, a usual SSA step is executed considering only the stochastic reactions, the reaction set is updated and the next hybrid simulation step is executed.

The approximation of deterministic reactions by using stationary distributions can only be applied when these distributions exist. Further, the stationary distributions must be reached much faster compared to the next firing time of a stochastic reaction. These are strong restrictions for a hybrid simulator. Besides, calculating the stationary distributions empirically can be a computationally expensive process including many SSA steps of the deterministic reactions. Therefore, we also developed an alternative method for the hybrid ML-Rules simulator dealing with the deterministic reactions by integrating them directly, see Algorithm 6.3. In this case, the simulator does not require stationary distributions to exist. Since deterministic reactions are static, they only change the amounts of population-based species and therefore, it is straightforward to convert them into ODEs and applying a numerical integration method to calculate them. Similar to the "probability of no reaction" presented by Haseltine and Rawlings [71], we restrict the leap size by the first $r$-quantile $Q(r)$ depending on the propensity sum of the deterministic reactions. This restriction can be beneficial, because propensities of stochastic reactions are not updated during the integration of the deterministic

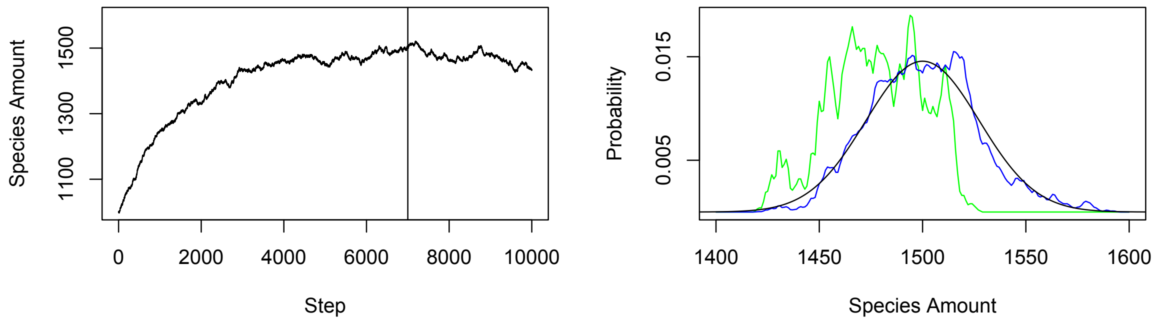



Figure 6.10: The shown results refer to a simple diffusion model with two rule schemes describing the movement of a species into and out of a cell. Left: A steady state estimator estimates the start of a steady state of the diffusing species at step 7000. Right: The approximated stationary distribution of the diffusing species after 10000 steps (green) and 100000 steps (blue) compared to the analytical result (black).

reactions and therefore, an error is induced. However, due to arbitrary complex rate equations and functions on solutions in ML-Rules, predicting this error is not possible in general and therefore we use a generic leap restriction not depending on error approximations. By default, we set $r = 0.01$. As shown in Algorithm 6.3 (ll.26-30), $r$ is increased or decreased depending on the relative change of the propensity sum of the stochastic reactions compared to a user-defined parameter $\epsilon \in [0, 1]$.

### 6.3.3 Results with a Benchmark Model

We tested the presented hybrid simulator for ML-Rules with a simple multi-level benchmark model, see Figure 6.11 and achieved a significant speed-up. We used three configurations of the hybrid simulator:

(1) Reaction partitioning without considering propensities and direct integration of deterministic reactions.

(2) Reaction partitioning considering propensities and direct integration of deterministic reactions.

(3) Reaction partitioning considering propensities and approximation of deterministic reactions with empirical stationary distributions.

We do not use the reaction partitioning without considering propensities and the approximation of deterministic reactions with empirical stationary distributions, because the species `C` does not achieve stationary distributions fast enough, so that this

**Algorithm 6.3** Sketch of the hybrid simulator for ML-Rules that integrates deterministic reactions.
$t_j$: current simulation time after $j$ simulation steps,
$R^d = \{R^d_1, \ldots, R^d_{|R^d|}\}$: deterministic reaction set,
$R^s = \{R^s_1, \ldots, R^s_{|R^s|}\}$: stochastic reaction set,
$a^s_1, \ldots, a^s_{|R^s|}$: propensities of stochastic reactions,
$\tilde{X}(t_j)$: intermediate state of the system after integrating the deterministic reactions and before executing a stochastic reaction.

```
1  // Update reaction sets, calculate propensities
2  initialize();
3  // Calculate the propensity sum of all stochastic reactions
4  a_0^s(X(t_j)) := sum_{i=1}^{|R^s|} a_i^s(X(t_j))
5  // Select a dynamic reaction to be executed analog to the reaction
6  // selection of the SSA (see Algorithm 4.1)
7  i := select(R^s)
8  // Sample the execution time τ of the next dynamic reaction
9  τ := Exp(a_0^s(X(t_j)))
10 // Compute the r-quantile Q(r) (r ∈ [0,1]) of this
11 // exponential distribution.
12 Q(r) := -ln(1-r) / a_0^s(X(t_j))
13 if (Q(r) < τ) {
14   // Integrate all static reactions until t_j + Q(r)
15   // No dynamic reaction is fired
16   X̃(t_j) := integrate(X(t_j), R^s, Q(r))
17   t_{j+1} := t_j + Q(r)
18 } else {
19   // Integrate the static reactions until t_j + τ
20   X̃(t_j) := integrate(X(t_j), R^s, τ)
21   t_{j+1} := t_j + τ
22 }
23 // Propensity sum of dynamic reactions with X̃(t_j)
24 a_0^d(X̃(t_j)) := sum_{i=1}^{|R^d|} a_i^d(X̃(t_j))
25 // Depending on the relative propensity change, adapt r
26 if (|1 - a_0^d(X(t_j))/a_0^d(X̃(t_j))| < ε) {
27   r := min(0.0001,r/2)
28 } else {
29   r := max(1,r·2)
30 }
31 // Execute the selected dynamic reaction
32 execute(X̃(t_j), R_i^d)
```

configuration of the hybrid simulator would degrade to the SSA. However, simulation results with the other three configurations are shown in Figure 6.12. The exact stochastic ML-Rules simulator needed $\approx 3$ hours to simulate one simulation run until

```
1  // parameter
2  r1 : 100.0; r2 : 0.001;
3
4  // entity definitions
5  A(); B(); C(); Context(bool)[];
6
7  // initial solution
8  >>INIT[1 Context(true)[2000 A + 500 B + 10 C] + 1000 A];
9
10 // 'fast' rules
11 Context(x)[A:a + s?] -> Context(x)[s?] + A @#a*r1;
12 Context(x)[s?] + A:a -> Context(x)[A + s?] @#a*r1;
13 Context(x)[A:a + s?] -> Context(x)[B + s?] @#a*r1;
14 B:b -> A @#b*r1;
15 // 'slow' rules
16 Context(x)[B:b + s?] -> Context(x)[C + s?] @#b*r2;
17 C:c -> @#c*r2;
18 Context(false)[s?] -> Context(true)[s?] @ r2;
19 Context(true)[s?] -> Context(false)[s?] @ r2;
```

Figure 6.11: A simple benchmark model to test the hybrid ML-Rules simulator.

simulation time 1000. The hybrid simulator is much faster with all configurations ((1) $\approx$ 1s, (2) $\approx$ 1s, (3) $\approx$ 20s) without a noticeable loss of accuracy. Configuration (3) is slower because it must calculate the stationary distribution for the species `A` and `B` after each execution of a stochastic reaction. Since the stationary distributions are only approximated, the noise of the species `A` and `B` is higher compared to the other configurations of the hybrid simulator (configuration (1) has no noise since all reactions changing `A`, `B`, and `C` are deterministic in this case).

### 6.3.4 Results with a Dictyostelium Discoideum Model

Besides experiments with the simple benchmark model, we executed experiments with a complex model representing the Dictyostelium discoideum amoeba aggregation process [12, p.95ff], see Appendix A.5. These amoebas are unicellular eukaryotic cells, which build multicellular slugs during their life cycles [23]. Since ML-Rules does not support spatial models explicitly, a grid-based space representation is applied. With the stochastic ML-Rules simulator, it was not possible to execute a useful simulation study for this model, since it needed more than 60 hours to simulate one amoeba cell until simulation time 5000. Although $\tau$-leaping performed better and only needs $\approx$ 1 minute to simulate one amoeba cell until simulation time 5000, it has been still to slow for more cells. However, to observe an aggregation process, many cells have to be simulated.

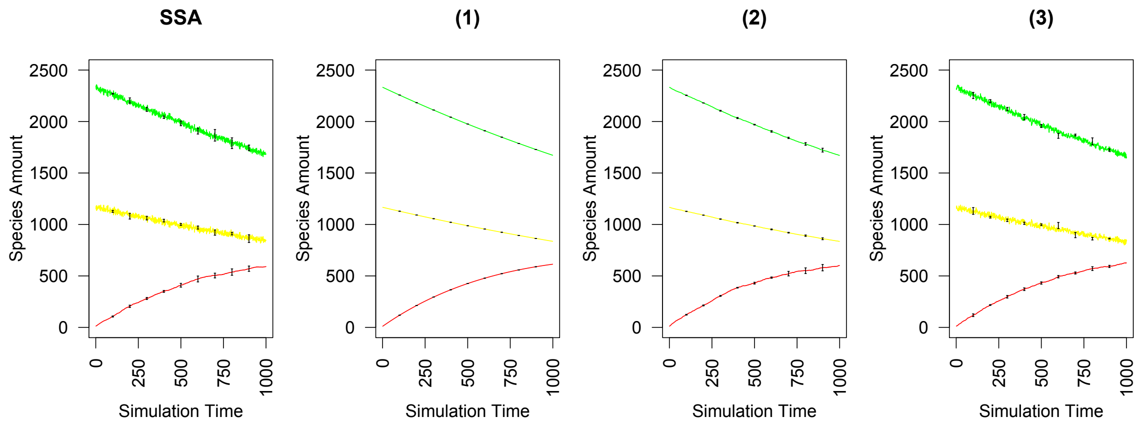



Figure 6.12: Comparison of simulation results (green: species A, yellow: species B, red: species C) of the exact stochastic ML-Rules simulator (left) and the hybrid simulator for ML-Rules with the configurations (1), (2), and (3) for the benchmark model described in Figure 6.11.

For experiments with more cells, we applied the hybrid ML-Rules simulator with a reaction partitioning not considering propensities and a direct integration of deterministic reactions. Referring to this model, considering propensities for the partitioning would result in the same partitions, since the stochastic reactions of this model are also the slow reactions. Further, we did not use the stationary distribution approximation, since the deterministic reactions do not induce stationary distributions.

However, the hybrid simulator with the suitable configuration achieved much better results than the stochastic simulator, see Figure 6.13 (top); it took $\approx 6$ hours to simulate 400 cells. Mainly, the benefit comes from a deterministic calculation of cell internal reactions, see Figure 6.13 (middle). The oscillatory behavior fits to the data computed in [107]. Further, the aggregation process of 400 Dictyostelium discoideum amoebas in a $20 \times 20$ grid is illustrated in Figure 6.13 (bottom). At the beginning of the simulation, all amoebas are equally distributed. The multicellular aggregates become larger over time, i.e., at $t = 5000$, most of the 400 amoebas are gathered at few points of the grid.

The results with the hybrid simulator are promising, nevertheless, much more amoebas would have to be simulated to study the aggregation process more comprehensively. The performance of the hybrid simulator is slowed down by many stochastic events representing moves of cells. In the next Section 6.3.5, we present a recent extension of the hybrid simulator to tackle this issue. However, besides runtime issues, for the first time with ML-Rules we also faced memory issues when simulating more than 1000 cells — the simulation aborted with an `OutOfMemory` exception. Memory improvements are therefore also in the focus of further improvements.

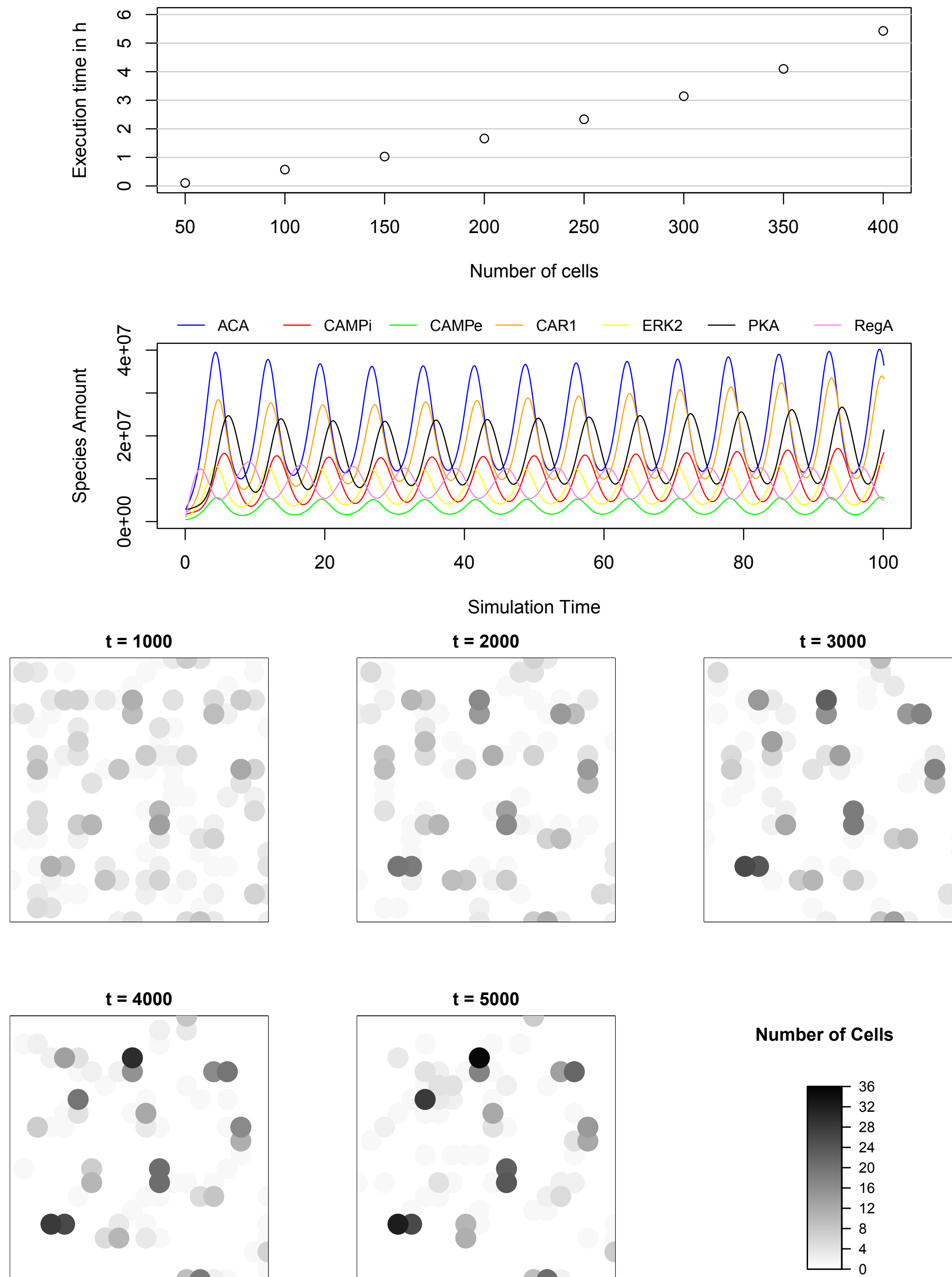



Figure 6.13: Top: Runtime to simulate dictyostelium discoideum amoebas (based on 20 replications). Middle: Oscillating amounts of cell internal species of an individual amoeba (calculated continuously). Bottom: The aggregation process of 400 amoebas in a $20 \times 20$ grid.

### 6.3.5 Parallel Execution of Stochastic Reactions

When executing the experiments with the Dictyostelium discoideum amoebas model, many stochastic reactions slow down the execution of a simulation run. As these stochastic reactions refer to movements of amoebas, they are often independent from each other, i.e., one amoeba can move without invalidating possible moves of many other amoebas. This observation motivates the extension of the hybrid ML-Rules simulator to not only perform leaps on deterministic reactions, but also on stochastic reactions, see [171, 198][4].

Algorithm 6.4 illustrates the hybrid simulator with multiple stochastic reaction firings per leap. Every stochastic reaction is at most fired once during one leap. This restriction is necessary since stochastic reactions usually cannot be fired multiple times. For example, stochastic reactions often change, remove or create compartments, i.e., after firing such a reaction, the reactant compartments are not available anymore and no reactions referring to the product compartments exist. The set of stochastic reactions to be fired is determined by repeating the usual SSA reaction selection process $|R^s|$ times (ll.6-13). Using this mechanism account for the propensities of each stochastic reaction. Further, a selected reaction $R_i^s$ is only added to the set of selected reactions $S$ if it can still be fired assuming all reactions in $S$ are fired, i.e., it does not share compartment reactants with any reaction in $S$. Thus, assuming independent stochastic reactions, in principle it is possible that all stochastic reactions are executed during a leap. On the other hand, at least one stochastic reaction is definitely selected. The time advance $\tau$ is sampled from an Erlang distribution with rate $a_0^s(X(t_j))$ and shape $|S|$. The Erlang distribution (`Erlang(`$\lambda$`,`$n$`)`) is a continuous distribution that can be used to sample the time of $n$ independent exponentially distributed events with rate $\lambda$. So far, this extension of the hybrid simulator has been tested with some simple benchmarks and will be part of future research.

## 6.4 Summary

The efficiency of the `Adaptive Simulator` depends on the available pool of simulators. Therefore, in this chapter, we have developed different tailored and approximate simulators for the modeling language ML-Rules to explore their efficiency and applicability to the `Adaptive Simulator`. We have chosen ML-Rules since it provides many computational challenges and it has potential for various tailored and approximate simulators.

Tailored simulators can exploit specific properties of models they are applicable to,

[4]In principle, this approach can also be applied to $\tau$-leaping.

**Algorithm 6.4** Sketch of a hybrid simulator for ML-Rules with multiple dynamic reaction firings per leap.
$t_j$: current simulation time after $j$ simulation steps,
$R^s = \{R^d_1, \ldots, R^d_{|R^d|}\}$: deterministic reaction set,
$R^s = \{R^s_1, \ldots, R^s_{|R^s|}\}$: stochastic reaction set,
$S$: set of selected stochastic reactions to be executed,
$a^s_1 \ldots a^s_{m_s}$: propensities of stochastic reactions,
$\tilde{X}(t_j)$: intermediate state of the system after integrating the deterministic reactions and before executing stochastic reactions.

```
1  // Update reaction sets, calculate propensities
2  initialize();
3  // Calculate the propensity sum of all stochastic reactions
4  a_0^s(X(t_j)) := Σ_{i=1}^{|R^s|} a_i^s(X(t_j))
5  // Select stochastic reactions to be executed
6  repeat |R^s| times {
7    // Select a stochastic reaction analog to the reaction
8    // selection of the SSA (see Algorithm 4.1)
9    i := select(R^s)
10   if (R_i^s ∉ S && isPossible(R_i^s,S)) {
11     S := S ∪ {R_i^s}
12   }
13 }
14 // Sample the execution time τ for |S| stochastic reaction from an
15 // Erlang distribtion with rate a_0^s(X(t_j)) and shape |S|
16 τ := Erlang(a_0^s(X(t_j)), |S|)
17 // Integrate deterministic reactions until t_j + τ
18 X̃(t_j) := integrate(X(t_j), R^d, τ);
19 // Execute the selected stochastic reaction (order does not
20 // matter as they must be independent from each other)
21 for (R_i^s ∈ S) {
22   execute(X̃(t_j), R_i^s)
23 }
24 t_{j+1} := t_j + τ
```

e.g., applying additional auxiliary data structures or avoiding unnecessary calculations. To decide whether a tailored simulator is applicable to a model, either explicit conditions are needed or automatic validity checks have to be applied, see Figure 6.1. For example, an automatic validity check could simulate a model until a specific simulation time with all available simulators and compare their results to decide which simulators are valid for the given model. Although this approach is generic, it requires different simulators to be available and it cannot guarantee the validity of a simulator, but it only increases the confidence that a simulator is probably applicable to a model.

For ML-Rules, we have developed two tailored simulators. In Section 6.1.1, we

present the `StaticSimulator` tailored to ML-Rules models with a fixed reaction set. For these models, the rule instantiation process has only to be done once at the beginning of a simulation run and afterward the simulator can behave like a usual SSA. For this simulator, we present explicit conditions, e.g., functions on solutions are not supported, that can be checked automatically via a static model analysis. In Section 6.1.2, we present the `LinkSimulator` tailored to ML-Rules models focusing on species bonds. This simulator improves the rule instantiation process for reactions between bound entities as follows. Typically, such rules consist of two reactants which share a unique link variable, see Figure 6.3 page 134 (l.11) . After matching a bound entity to the first reactant of such a rule, the matching of the second reactant is clearly determined, i.e., these rules fulfill the *rigidity property*, see Section 6.1 page 131. Consequently, instead of testing a set of entities for the second reactant to find the unique match, the only valid match can be selected directly. The conditions for applying this simulator are less restrictive compared to the conditions for the `StaticSimulator`, e.g., functions on solutions not changing any link attributes are still allowed, and the conditions can still be checked automatically via static model analysis. Altogether, both simulators make it possible to simulate models applicable to them more efficiently compared to the basic ML-Rules simulator. Further, for each model, it can automatically be checked whether these simulators can be applied and finally, they also produce the same results as the exact basic ML-Rules simulator. Consequently, they can be applied to the `Adaptive Simulator` straightforwardly to improve its efficiency.

In contrast to tailored simulators, approximate simulators trade accuracy for speed and therefore change the quality of simulation results. However, typically they achieve a significant speed-up with an acceptable loss of accuracy and are therefore a valuable approach to improve the efficiency of simulation runs. By developing approximate simulators for ML-Rules, i.e., $\tau$-leaping (see Section 6.2) and a hybrid simulator (see Section 6.3), we demonstrate that also for ML-Rules approximate simulators can speed-up simulation runs significantly. Basically, both approximate simulators partition reactions based on structural changes, i.e., *static* reactions are approximated and other reactions are still calculated exactly. Such a separation of reactions is not unusual, e.g., the modeling formalism dynDEVS is explicitly separating transitions changing the structure to an additional *model transition function* [190]. Nevertheless, the developed approximate simulators are not exact anymore. Thus, accuracy analysis is necessary to evaluate the quality of simulation results. However, analyzing the accuracy of a simulator during runtime without an explicit error estimation method is challenging. All in all, when applying an approximate simulator together with the `Adaptive Simulator`, the action set must be restricted so that it only contains

different configurations of this simulator and further, no parameters and components of the simulator influencing the accuracy of the simulation results are allowed to be adapted. Otherwise, the `Adaptive Simulator` can influence the simulation results by performing adaptations violating one of its fundamental requirements, see Section 3.1.

So far, to analyze the accuracy of simulation results, we applied visual analytics to analyze the results of $\tau$-leaping for ML-Rules, however, this process has two drawbacks. First, it is applied after executing all simulation runs. Therefore, it is not helpful to analyze the accuracy of simulators *during* runtime. Second, it is by definition a technique processed manually by users; it cannot be automated.

# Chapter 7

# Conclusions and Outlook

> It's time to design and build computing systems capable of *running themselves*, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them. These autonomic systems must anticipate needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing systems to get them there.
>
> Paul Horn [92]

## 7.1 Summary

The complexity challenge pronounced by IBM in 2001 [92] has also reached the modeling and simulation community: Solutions are needed and developed to deal with complex models, complex simulation experiments and complex simulation software. Referring to simulation software, essential concepts of software engineering like abstraction, separation of concerns, reuse, and design patterns should be applied. An established approach following these concepts is a component-based software design, which has been also the basis of the modeling and simulation framework JAMES II [87]. Although this design helps to deal with complex software systems, it induces a configuration challenge due to compositional opportunities. This challenge should be tackled with adaptive methods inspired by concepts like self-adaptive software [105] or programming by optimization [91], resulting in methods that automatically select and change compositions and configurations.

In Chapter 2, we give an overview about various methods used in the area of modeling and simulation to adapt simulators. We determine and analyze these methods based on four properties, see Table 2.1 page 31:

1. Considered Features / Measurements: What features and measurements are used for the adaptation decisions (model properties, simulator properties, environment properties, performance).
2. Adapted Property: primitive parameters (e.g., thresholds), complex parameters (e.g., partitioning of LPs), simulator.
3. Trigger & Frequency: initialization (i.e., one adaptation during the initialization of a simulation run), interval, conditional.
4. Quality Change: Whether adaptations change the quality of simulation results.

Various kinds of features are used to control the adaptation process. Some methods rely on model properties, e.g., $\tau$-leaping is only using the model state to determine a suitable leap size [25]. Other methods rely on simulator properties, e.g., the number of rollbacks is considered by *Penalty-Based Throttling* to adapt the optimism of the LPs [163]. Environment properties like the CPU load are also sometimes considered [18]. The *Supervised Simulator Selection* and the *Unsupervised Simulator Selection* consider the runtime performance of the simulator [47].

Besides, an adaptation itself is often simple to be executed, i.e., mostly primitive parameters like thresholds, delays, or the step size are adapted. The adaptation frequency is often fixed and user-defined, i.e., a fixed adaptation interval depending on the number of executed simulation events is applied. Some approaches apply only one adaptation that is executed during the initialization of a simulation run. In case the quality of results can be changed by adaptations, the calculated error is always considered in some way, e.g., by comparing results of two simulator configurations with different accuracy, see Section 2.2. Considering the accuracy of results is necessary, because otherwise the simulator could trade too much accuracy to achieve a better runtime performance eventually making the results useless.

Altogether, the methods analyzed in Chapter 2 emphasize that adapting simulators can be beneficial. Nevertheless, most of the presented methods are tailored to a specific application scenario and cannot be reused straightforwardly for other scenarios. Further, existing generic methods like the *Unsupervised Simulator Selection* [47] do not adapt simulators during the execution of a simulation run, but they select a simulator once during the initialization of a simulation run.

Motivated by our conclusions about methods adapting simulators, in Chapter 3 we present established approaches to categorize and analyze adaptive software and map these to the presented methods adapting simulators. Thereby, we refer to the facets presented by Anderson et al. to analyze and evaluate adaptive software: goals, changes, mechanisms, and effects [3].

Besides, we refine the notion of dynamic adaptations [135], i.e., adaptations calculated at runtime, to *weak dynamic adaptations* and *strong dynamic adaptations*, see Figure 3.2 page 37. Whereas *weak dynamic adaptations* are calculated during runtime of the simulation software, but not within the runtime of a simulation run, *strong dynamic adaptations* also allow adapting a simulator during the execution of a simulation run. Further, we emphasize the distinction between *parameter adaptation* and *compositional adaptation*. Parameter adaptations refer to methods mainly using fixed and specific adaptation trigger, adaptation options and decision making processes. In contrast, compositional adaptations are more flexible typically using machine learning, e.g., to deal with a dynamic set of adaptation options. Altogether, most presented methods adapting simulators perform strong dynamic adaptations with parameter adaptations. Only the *Supervised Simulator Selection* and the *Unsupervised Simulator Selection* perform compositional adaptations [47]. Nevertheless, these two methods perform weak dynamic adaptations, i.e., they select a simulator during the initialization of a simulation run. To realize compositional adaptations — from a technological viewpoint — separation of concerns, component-based design, and computational reflection are key characteristics that should be considered [135]. Further, an explicit decision making process should be separated from the business logic [30].

Following the given discussion, we motivate a generic adaptation method performing strong dynamic adaptations as well as compositional adaptations, which is developed in Chapter 4: the `Adaptive Simulator`. It performs adaptations during the execution of a simulation run, adaptations can change the structure of the simulator, and it does not use a fixed set of adaptation options and a predefined adaptation strategy, but it uses reinforcement learning to learn autonomously which adaptations to perform.

Based on the identified essential requirements for the `Adaptive Simulator` in Section 4.1, we developed the structure of the `Adaptive Simulator` as follows. Firstly, we integrate the `Adaptive Simulator` into the component-based modeling and simulation framework JAMES II and let it implement the same interface `IProcessor` that has to be implemented by all simulators in JAMES II. However, the `Adaptive Simulator` does not calculate simulation events directly, but it uses the wrapper pattern [60] to encapsulate available simulators applicable to a specific problem, adapts the currently used "internal simulator" as needed and it employs reinforcement learning [185] to explore and exploit the performance of these simulators. Specifically, Q-learning [14] is applied saving the utility based on the event throughput of each state-action pair in a q-value matrix. The encapsulated simulators are used to calculate the state transitions of a model. Thus, the `Adaptive Simulator` is not restricted to any modeling language, but it can be applied to all modeling approaches available

in JAMES II. Thereby, a clear separation between the business logic, i.e., the execution of the state transitions, and the decision making process is achieved. Further, by using the `Registry` of JAMES II, the `Adaptive Simulator` computes the set of available simulators to proceed with the simulation run automatically. The `Adaptive Simulator` itself is realized as a component-based simulator — all important concerns (e.g., adaptation trigger and the reward function) of the `Adaptive Simulator` are separated into individual components making it flexible to integrate new methods and algorithms.

When executing an adaptation, the `Adaptive Simulator` selects a new internal simulator to proceed with the model execution that is exchanged completely with the previous internal simulator. The new internal simulator simply uses the current state of the model to initialize itself properly. Thus, there is no need to determine differences between the old and the new internal simulator and no data structures must be checked and updated to guarantee the integrity of the new internal simulator.

We introduce *base states* ($\sigma \in \Sigma$) to represent all available information about the model, the simulator and the environment that can be collected by the `Adaptive Simulator` after each event execution. Since adaptations shall not be executed per default after each event execution, all base states between two adaptations are combined to a *base state trajectory* ($\tau \in \Sigma^*$). Finally, a base state trajectory is transformed to a state $s \in S$ used by the reinforcement learning method.

High-dimensional or infinite state spaces can be challenging as they reduce the learning efficiency. We integrated three different generalization methods into the `Adaptive Simulator` to deal with this issue, see Section 4.3: 1) a grid-based generalization method, 2) the `Decision Boundary Partitioning Algorithm` (DBPA) [165], and 3) the `Adaptive Vector Quantization` (AVQ) [114]. These methods generalize a state $s \in S$ to a macro state $m \in M$ representing an area of the state space, whereby $M \subseteq S \wedge |M| \ll |S|$. In the ideal case, each area represented by a macro state has a homogeneous performance behavior of all states within this area.

Referring to adaptation trigger, we present three approaches, see Section 4.4: 1) using a fixed adaptation condition based on the wallclock time, simulation time and number of processed events, 2) using a set of conditions integrated into the adaptation actions, and 3) using the event throughput to apply a changepoint detection method [1]. We did not focus on model specific adaptation trigger, e.g., to trigger an adaptation after the execution of a rare event, since this would contradict the generality of the `Adaptive Simulator`.

We evaluate various aspects of the `Adaptive Simulator` in Chapter 5. Firstly, we motivate a component-based simulator for the modeling language ML-Rules [130] by discussing its computational challenges mainly induced by dynamic reaction networks,

attributed species, and functions on solutions. This component-based simulator results in manifold configuration possibilities making it a suitable candidate to evaluate the `Adaptive Simulator`. We illustrate the effectiveness of the `Adaptive Simulator` by using an ML-Rules benchmark model with two different phases. Specifically, we analyze the impact of different multi-armed bandit policies for the action selection. Although being simple, the $\epsilon$-decreasing policy has outperformed the other policies. Further, we apply the three developed state space generalizations together with the `Adaptive Simulator` and the ML-Rules benchmark model. Fixed grids can perform well, however, as expected their effectiveness essentially depends on the chosen grid size. The DBPA usually outperformed fixed grids, but it sometimes failed resulting in a worse performance compared to fixed grids. Further, it has already been challenging to determine a suitable configuration of the DBPA. The AVQ has been more robust, but never performed as good as the DBPA. Referring to the changepoint detection method, it proved to be robust and effective. In contrast, the performance of fixed adaptation conditions depends on the chosen conditions, which are not obvious to choose.

Besides the ML-Rules benchmark model, we executed experiments also with more complex models used in simulations studies: a Cell Cycle model [130], an Endocytosis model [74], and a Wnt/$\beta$-catenin pathway model [131, 132, 69]. For these, the `Adaptive Simulator` has been able to detect the best performing configuration of the component-based ML-Rules simulator, but it could not outperform it. The computational demands might not differ sufficiently to make runtime adaptations beneficial. However, detecting the best-performing simulator is also a challenging task the `Adaptive Simulator` has successfully solved making it beneficial compared to the random choice of a simulator. Further, since the `Adaptive Simulator` already exploits its knowledge within one replication, it also outperforms the `AdaptiveSimulationRunner` [47] in case few replications are executed. Besides, by successfully applying the `Adaptive Simulator` to the modeling languages SR [97] and PDEVS [205], we emphasize its flexibility and that it is not restricted to any modeling language available in JAMES II.

In Chapter 6, we present tailored and approximate simulators for ML-Rules to support the effectiveness of the `Adaptive Simulator`. Due to its various computational challenges, ML-Rules is a suitable candidate to develop such simulators. Tailored simulators only allow simulating a subset of ML-Rules models, thereby exploiting specific properties of these models. We present a specific simulator (`StaticSimulator`) only supporting ML-Rules models with *fixed* reaction networks and achieved a significant speed-up with this simulator. Further, we present a specific simulator (`LinkSimulator`) focusing on ML-Rules models with species bonds and also achieved a significant speed-

up with this simulator. If applicable, both simulators still produce exact results referring to the basic ML-Rules simulator. Further, since we define exact conditions to check whether they are applicable to a model or not, in principle they could directly be applied to the `Adaptive Simulator`.

In contrast to tailored simulators, approximate simulators trade accuracy for speed and therefore change the quality of simulation results. However, typically they achieve a significant speed-up with an acceptable loss of accuracy and are therefore a valuable approach to improve the efficiency of simulation runs. We demonstrate the potential of approximate simulators by developing a $\tau$-leaping simulator [25] and a hybrid simulator [42] for ML-Rules. Both simulators partition the reaction set based on reaction network changes — a common approach to separate reactions [190]. Since the approximate simulators do not calculate exact results anymore, accuracy analysis is necessary to evaluate their quality. However, due to the expressiveness of ML-Rules, an explicit error estimation method cannot be defined for the developed approximate simulators. As a starting point, we successfully applied a visual analytics tool [123] to evaluate the accuracy of results calculated by different $\tau$-leaping configurations. Nevertheless, this analysis is done after executing all simulation runs. Therefore, since we do not have developed an alternative method to measure the accuracy of the approximate simulators during runtime yet, these simulators cannot simply be applied to the `Adaptive Simulator`. Instead, the action set must be restricted so that it only contains simulators calculating results with the same accuracy and the accuracy is not allowed to be influenced by adaptations. Otherwise, the `Adaptive Simulator` could tend to trade more and more accuracy making the simulation results useless.

## 7.2 Outlook

The `Adaptive Simulator` shows that strong dynamic adaptations and compositional adaptations can be successfully combined for component-based simulation systems. However, there is room for improvements and extensions. So far, the `Adaptive Simulator` has not been applied to parallel and distributed discrete event simulation. It could be used locally for each logical process to adapt its behavior or as a central decision maker changing the global simulator type. Further, to deal with large action sets, methods to create simulator portfolios at runtime should be developed and applied, e.g., see [202]. Since the `Adaptive Simulator` consists of various components with parameters, configuring it is an important issue that should be tackled, e.g., by using Meta-learning techniques [194].

A complex challenge still lies in the feature selection for the creation of base states. Currently, we delegate this challenge to the developer of the components used

by the `Adaptive Simulator`. However, these developers might not be aware of the existence of the `Adaptive Simulator` or of important features of their components, i.e., important features might not be forwarded to the `Adaptive Simulator`. Further, we have not considered environment properties explicitly so far. Here, generic methods should be explored solving the feature selection problem more autonomously.

The specialized ML-Rules simulators achieved significant speed-ups compared to the standard ML-Rules simulator. Since they still compute exact results and we defined clear conditions to apply these simulators, they can be applied by the `Adaptive Simulator` straightforwardly. However, the conditions have been set restrictively, i.e., models might be rejected although they are applicable to these simulators. More sophisticated static model analysis methods for ML-Rules are needed to improve the validity checks. Moreover, it should be explored whether further specialized simulators can be developed for ML-Rules, e.g., based on existing simulators for similar modeling languages like NFSim for BioNetGen [179].

Referring to the approximate simulators developed for ML-Rules, future work should focus on methods to measure the accuracy of these approximate simulators at runtime. Only with explicit accuracy measurements, the approximate simulators can be compared automatically and the `Adaptive Simulator` can exploit the methods by considering their accuracy to control adaptations.

# Bibliography

[1] R. P. Adams and David. J. C. MacKay. Bayesian Online Changepoint Detection. Technical report, University of Cambridge, 2007.

[2] OSGi Alliance. *OSGi Service Platform: Core Specification*. aQute Publishing, 2009.

[3] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems*, volume 5525, pages 27–47. 2009.

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented Programming with Java. *Information and Media Technologies*, 6(2):399–419, 2011.

[5] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, pages 135–173. Springer Berlin Heidelberg, 2006.

[6] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2–3):235–256, May 2002.

[7] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E Schapire. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331, October 1995.

[8] Diana M Ball and Stephen M Hoyt. The adaptive Time-Warp concurrency control algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 174–177, January 1990.

[9] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.

[10] Andrew G Barto and Sridhar Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379, October 2003.

[11] Pavol Bauer, Jonatan Lindén, Stefan Engblom, and Bengt Jonsson. Efficient Inter-Process Synchronization for Parallel Discrete Event Simulation on Multicores. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (PADS)*, pages 183–194, 2015.

[12] Marco Beccuti, Mary Ann Blätke, Martin Falk, Simon Hardy, Monika Heiner, Carsten Maus, Sebastian Nähring, and Christian Rohr. Dictyostelium discoideum: Aggregation and Synchronisation of Amoebas in Time and Space. *Dagstuhl Reports: Multiscale Spatial Computational Systems Biology (Dagstuhl Seminar 14481)*, 4(11):195–214, 2015.

[13] R Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6:679–684, 1957.

[14] R. Bellman and Rand Corporation. *Dynamic Programming*. Princeton University Press, 1957.

[15] Nelly Bencomo and Amel Belaggoun. Supporting decision-making for self-adaptive systems: from goal models to dynamic decision networks. In *Requirements Engineering: Foundation for Software Quality*, pages 221–236. Springer, 2013.

[16] Arne T. Bittig, Florian Reinhardt, Simone Baltrusch, and Adelinde M. Uhrmacher. Predictive Modelling of Mitochondrial Spatial Structure and Health. In *Proceedings of the 12th International Conference on Computational Methods in Systems Biology, CMSB 2014*, pages 252–255, 2014.

[17] Michael L. Blinov, James R. Faeder, Byron Goldstein, and William S. Hlavacek. BioNetGen: Software for Rule-based Modeling of Signal Transduction Based on the Interactions of Molecular Domains. *Bioinformatics*, 20(17):3289–3291, 2004.

[18] Azzedine Boukerche and Sajal K Das. Dynamic load balancing strategies for conservative parallel simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 20–28, July 1997.

[19] R. Brown. Calendar Queues: A Fast 0(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[20] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. A Conceptual Framework for Adaptation. In *Fundamental Approaches to Software Engineering*, volume 7212, pages 240–254. 2012.

[21] Peter Bunus. A Simulation and Decision Framework for Selection of Numerical Solvers in Scientific Computing. In *Proceedings of the 39th Annual Simulation Symposium (ANSS)*, pages 178–187, 2006.

[22] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, volume 1 edition, 1996.

[23] Daniel S. Calovi, Leonardo G. Brunnet, and Rita M. C. de Almeida. cAMP diffusion in *Dictyostelium discoideum*: A Green's function method. *Physical Review E*, 82:011909, 2010.

[24] Yang Cao, Daniel T Gillespie, and Linda R Petzold. The slow-scale stochastic simulation algorithm. *The Journal of Chemical Physics*, 122(1), 2005.

[25] Yang Cao, Daniel T Gillespie, and Linda R Petzold. Efficient step size selection for the tau-leaping simulation method. *The Journal of Chemical Physics*, 124(4), January 2006.

[26] Yang Cao, Daniel T. Gillespie, and Linda R. Petzold. Adaptive explicit-implicit tau-leaping method with automatic tau selection. *The Journal of Chemical Physics*, 126(22), 2007.

[27] Yang Cao, Hong Li, and Linda Petzold. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *The Journal of Chemical Physics*, 121(9):4059–4067, September 2004.

[28] François E Cellier. *Continuous System Modeling*. Springer, 1991.

[29] K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *Software Engineering, IEEE Transactions on*, SE-5(5):440–452, 1979.

[30] Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller,

Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525, pages 1–26. 2009.

[31] Ryan Child and Philip Wilsey. Dynamically Adjusting Core Frequencies to Accelerate Time Warp Simulations in Many-Core Processors. In *Proceedings of the 26th Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 35–43, 2012.

[32] Petra Claeys, Filip Claeys, Bernard De Baets, and Peter A Vanrolleghem. Intelligent configuration of numerical solvers of environmental ODE/DAE models using machine learning techniques. In *Proceedings of the International Congress on Environmental Modelling and Software (iEMSs)*, 2006.

[33] Petra Claeys, Peter A Vanrolleghem, and Bernard De Baets. Automatic numerical solver selection from a repository of pre-run simulations. *Water science and technology : a journal of the International Association on Water Pollution Research*, 59(5):893—–906, 2009.

[34] Zack Coker, David Garlan, and Claire Le Goues. SASS: Self-adaptation using stochastic search. In *Proceedings 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, 2015.

[35] Alina Crudu, Arnaud Debussche, and Ovidiu Radulescu. Hybrid stochastic simplifications for multiscale gene networks. *BMC Systems Biology*, 3(89), 2009.

[36] Vincent Danos, Jérôme Feret, Walter Fontana, and Jean Krivine. Scalable Simulation of Cellular Signaling Networks. In *Proceedings of the 5th Asian Symposium on Programming Languages and Systems, APLAS 2007*, pages 139–157, Berlin, Heidelberg, 2007. Springer-Verlag.

[37] Vincent Danos and Cosimo Laneve. Formal molecular biology. *Theoretical Computer Science*, 325(1):69–110, 2004.

[38] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A Survey of Autonomic Communications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 1(2):223–259, 2006.

[39] J.R. Dormand and P.J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26, 1980.

[40] Weinan E, Di Liu, and Eric Vanden-Eijnden. Nested stochastic simulation algorithm for chemical kinetic systems with disparate rates. *The Journal of Chemical Physics*, 123(19), 2005.

[41] Thomas Eiter and Heikki Mannila. Computing Discrete Fréchet Distance. Technical Report CD-TR 94/64, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria, 1994.

[42] J. Elf and M. Ehrenberg. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. *Systems Biology*, 1(2):230–236, 2004.

[43] S. Endrikat and S. Hanenberg. Is Aspect-Oriented Programming a Rewarding Investment into Future Code Changes? A Socio-technical Study on Development and Maintenance Time. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*, pages 51–60, 2011.

[44] Robert Engelke and Roland Ewald. Configuring Simulation Algorithms with ParamILS. In *Proceedings of the Winter Simulation Conference*, pages 391:1–391:2, 2012.

[45] Thomas Erl. *Service-oriented architecture: concepts, technology, and design.* Pearson Education India, 2005.

[46] Erik Ernst. Separation of concerns. In *Proceedings of the AOSD 2003 Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT)*, 2003.

[47] Roland Ewald. *Automatic Algorithm Selection for Complex Simulation Problems.* PhD thesis, University of Rostock, Germany, 2010.

[48] Roland Ewald, J Himmelspach, M Jeschke, S Leye, and A M Uhrmacher. Flexible experimentation in the modeling and simulation framework JAMES II—implications for computational systems biology. *Briefings in Bioinformatics*, 11(3):290–300, 2010.

[49] Roland Ewald, Jan Himmelspach, and Adelinde M Uhrmacher. An Algorithm Selection Approach for Simulation Systems. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS'08)*, pages 91–98, 2008.

[50] Roland Ewald, Stefan Leye, and Adelinde M Uhrmacher. An Efficient and Adaptive Mechanism for Parallel Simulation Replication. In *Proceedings of the 23rd Workshop on Principles of Advanced and DistributedSimulation (PADS'09)*, pages 104–113, 2009.

[51] Roland Ewald, R Schulz, and A M Uhrmacher. Selecting simulation algorithm portfolios by genetic algorithms. In *Proceedings of the 24th Workshop on Principles of Advanced and Distributed Simulation (PADS'10)*, pages 1–9, 2010.

[52] Roland Ewald and A M Uhrmacher. Automating the runtime performance evaluation of simulation algorithms. In *Proceedings of the 41st Winder Simulation Conference (WSC'09)*, pages 1079–1091, 2009.

[53] Roland Ewald and Adelinde M. Uhrmacher. SESSL: A Domain-specific Language for Simulation Experiments. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 24(2):11:1–11:25, 2014.

[54] A. Falcon, P. Faraboschi, and D. Ortega. An Adaptive Synchronization Technique for Parallel Simulation of Networked Clusters. In *International Symposium on Performance Analysis of Systems and software*, pages 22–31, 2008.

[55] Lars Ferm, Andreas Hellander, and Per Lötstedt. An adaptive algorithm for simulation of stochastic reaction–diffusion processes. *Journal of Computational Physics*, 229(2):343–360, 2010.

[56] Mark B. Flegg, S. Jonathan Chapman, and Radek Erban. The two-regime method for optimizing stochastic reaction–diffusion simulations. *Journal of The Royal Society Interface*, 9(70):859–868, 2012.

[57] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *Software, IEEE*, 23(2):62–70, 2006.

[58] Richard M Fujimoto. *Parallel and distributed simulation systems*, volume 300. Wiley New York, 2000.

[59] Matteo Gagliolo and Jürgen Schmidhuber. Learning Dynamic Algorithm Portfolios. *Annals of Mathematics and Artificial Intelligence*, 47:295–328, 2006.

[60] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Software Components*. Addison-Wesley Professional, 1994.

[61] Samik Ghosh, Yukiko Matsuoka, Yoshiyuki Asai, Kun-Yi Hsin, and Hiroaki Kitano. Software for systems biology: from tools to integrated platforms. *Nature Reviews Genetics*, 12:821–832, 2011.

[62] Michael A Gibson and Jehoshua Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *The Journal of Chemical Physics*, 104(9):1876–1889, February 2000.

[63] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[64] Daniel T Gillespie. Approximate accelerated stochastic simulation of chemically reacting system. *The Journal of Chemical Physics*, 115(4):1716–1733, July 2001.

[65] Rick Siow Mong Goh and Ian Li-Jin Thng. Mlist: An efficient pending event set structure for discrete event simulation. *International Journal of Simulation-Systems, Science & Technology*, 4(5-6):66–77, 2003.

[66] M.G. Gouda and T. Herman. Adaptive Programming. *IEEE Transactions on Software Engineering*, 17(9):911–921, 1991.

[67] Isabelle Guyon and André Elisseeff. An Introduction to Variable and Feature Selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[68] Fiete Haack, Kevin Burrage, Ronald Redmer, and Adelinde M Uhrmacher. Studying the role of lipid rafts on protein receptor bindings with Cellular Automata. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(3):760–770, 2013.

[69] Fiete Haack, Heiko Lemcke, Roland Ewald, Tareck Rharass, and Adelinde M. Uhrmacher. Spatio-temporal Model of Endogenous ROS and Raft-Dependent WNT/Beta-Catenin Signaling Driving Cell Fate Commitment in Human Neural Progenitor Cells. *PLoS Computational Biology*, 11(3):e1004106, 2015.

[70] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[71] Eric L. Haseltine and James B. Rawlings. Approximate simulation of coupled fast and slow reactions for stochastic chemical kinetics. *The Journal of Chemical Physics*, 117(15):6959–6969, 2002.

[72] T. Helms, C. Maus, F. Haack, and A. M. Uhrmacher. Multi-level modeling and simulation of cell biological systems with ML-Rules - A tutorial. In *Proceedings of the Winter Simulation Conference*, pages 177–191, 2014.

[73] Tobias Helms. *Inkrementelle Konstruktion von Algorithmenportfolios zur Simulation*. Bachelor's thesis, University of Rostock, 2011.

[74] Tobias Helms. *Adaptive Laufzeit-Konfiguration von ML-Rules Simulationen*. Master's thesis, University of Rostock, 2012.

[75] Tobias Helms, Roland Ewald, Stefan Rybacki, and Adelinde M Uhrmacher. A Generic Adaptive Simulation Algorithm for Component-based Simulation Systems. In *Proceedings of the 27th Workshop on Principles of Advanced and Distributed Simulation (PADS'13)*, pages 11–22, 2013.

[76] Tobias Helms, Roland Ewald, Stefan Rybacki, and Adelinde M. Uhrmacher. Automatic Runtime Adaptation for Component-Based Simulation Algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(1):7:1–7:24, 2015.

[77] Tobias Helms, Jan Himmelspach, Carsten Maus, Oliver Röwer, Johannes Schützel, and Adelinde M. Uhrmacher. Toward a Language for the Flexible Observation of Simulations. In *Proceedings of the Winter Simulation Conference*, pages 418:1–418:12, 2012.

[78] Tobias Helms, Martin Luboschik, Heidrun Schumann, and Adelinde M Uhrmacher. An Approximate Execution of Rule-Based Multi-level Models. In *Proceedings of the 11th International Conference on Computational Methods in Systems Biology (CMSB'13)*, pages 19–32, 2013.

[79] Tobias Helms, Steffen Mentel, and Adelinde M. Uhrmacher. Dynamic State Space Partitioning for Adaptive Simulation Algorithms. In *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'15, pages 149–152, 2016.

[80] Tobias Helms, Oliver Reinhardt, and Adelinde M. Uhrmacher. Bayesian Changepoint Detection for Generic Adaptive Simulation Algorithms. In *Proceedings of the 48th Annual Simulation Symposium*, ANSS '15, pages 62–69, 2015.

[81] Tobias Helms, Tom Warnke, Carsten Maus, and Adelinde M. Uhrmacher. Semantics and efficient simulation algorithms of an expressive multi-level modeling language. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*. accepted.

[82] Tobias Helms, Tom Warnke, and Adelinde M. Uhrmacher. Multi-level Modeling and Simulating of Cell Biological Systems - ML-Rules at Work. In *Methods in Molecular Biology*, 2016. submitted.

[83] Mostafa Herajy and Monika Heiner. Hybrid representation and simulation of stiff biochemical networks. *Nonlinear Analysis: Hybrid Systems*, 6(4):942–959, 2012.

[84] Jan Himmelspach. *Konzeption, Realisierung und Verwendung eines allgemeinen Modellierungs-, Simulations und Experimentiersystems*. PhD thesis, University of Rostock, Germany, 2007.

[85] Jan Himmelspach, Roland Ewald, Stefan Leye, and Adelinde M. Uhrmacher. Parallel and Distributed Simulation of Parallel DEVS Models. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, pages 249–256, 2007.

[86] Jan Himmelspach and Adelinde M Uhrmacher. Sequential processing of PDEVS models. In *Proceedings of the 3rd EMSS*, pages 239–244, 2006.

[87] Jan Himmelspach and Adelinde M. Uhrmacher. Plug'n simulate. In *Proceedings of the 40th Annual Simulation Symposium (ANSS'07)*, pages 137–143, 2007.

[88] Jan Himmelspach and Adelinde M. Uhrmacher. The Event Queue Problem and PDevs. In *Proceedings Spring Simulation Multiconference - Volume 1*, pages 257–264, 2007.

[89] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[90] Marcel Holle. *Dynamic State Space Representation for Adaptive Simulation Algorithms*. Bachelor's thesis, University of Rostock, 2013.

[91] Holger H. Hoos. Programming by Optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[92] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology, 2001.

[93] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275:51–54, 1997.

[94] Frank Hutter, Thomas Stützle, Kevin Leyton-Brown, and Holger H. Hoos. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[95] Walter L. Hürsch and Cristina Videira Lopes. Separation of Concerns. Technical report, Northeastern University, 1995.

[96] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[97] Matthias Jeschke and Roland Ewald. Large-Scale Design Space Exploration of SSA. In *Proeecdings of the 6th International Conference on Computational Methods in Systems Biology*, pages 211–230, 2008.

[98] Mathias John, Cédric Lhoussaine, Joachim Niehren, and Cristian Versari. Biochemical Reaction Rules with Constraints. In *Proceedings of the 20th European Symposium on Programming, ESOP*, pages 338–357, 2011.

[99] Simon L Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[100] Lucas N. Joppa, Greg McInerny, Richard Harper, Lara Salido, Kenji Takeda, Kenton O'Hara, David Gavaghan, and Stephen Emmott. Troubling Trends in Scientific Software Use. *Science*, 340(6134):814–815, 2013.

[101] Leslie P Kaelbling. *Learning in Embedded Systems*. MIT Press, 1993.

[102] M. S. Kamel, W. H. Enright, and K. S. Ma. ODEXPERT: An Expert System to Select Numerical Solvers for Initial Value ODE Systems. *ACM Transactions on Mathematical Software*, 19(1):44–62, 1993.

[103] Jonathan R. Karr, Jayodita C. Sanghvi, Derek N. Macklin, Miriam V. Gutschow, Jared M. Jacobs, Benjamin Bolival Jr., Nacyra Assad-Garcia, John I. Glass, and Markus W. Covert. A Whole-Cell Computational Model Predicts Phenotype from Genotype. *Cell*, 150(2):389–401, 2012.

[104] W David Kelton and Averill M Law. *Simulation modeling and analysis*. McGraw Hill Boston, 2nd edition, 1991.

[105] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[106] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP'97 — Object-Oriented Programming*, volume 1241, pages 220–242. 1997.

[107] Jongrae Kim, Pat Heslop-Harrison, Ian Postlethwaite, and Declan G Bates. Stochastic Noise and Synchronisation during *Dictyostelium* Aggregation Make cAMP Oscillations Robust. *PLoS Computational Biology*, 3(11):e218, 2007.

[108] Kenji Kira and Larry A. Rendell. The Feature Selection Problem: Traditional Methods and a New Algorithm. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*, pages 129–134, 1992.

[109] Donald E. Knuth. Big Omicron and Big Omega and Big Theta. *ACM SIGACT News*, 8(2):18–24, 1976.

[110] Granino Arthur Korn and John V Wait. *Digital continuous-system simulation*. Prentice Hall, 1978.

[111] Glenn E Krasner and Stephen T Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of object oriented programming*, 1(3):26–49, 1988.

[112] Thilo Krüger and Verena Wolf. *Hybrid Stochastic Simulation of Rule-Based Polymerization Models*, pages 39–53. 2016.

[113] Tomas G Kurtz. *Approximation of Population Processes*. SIAM, 1981.

[114] Ivan S.K. Lee and Henry Y.K. Lau. Adaptive state space partitioning for reinforcement learning. *Engineering Applications of Artificial Intelligence*, 17(6):577–588, 2004.

[115] M. Lees, B. Logan, Chen Dan, T. Oguara, and G. Theodoropoulos. Decision-theoretic throttling for optimistic simulations of multi-agent systems. In *9th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 171–178, 2005.

[116] Stefan Leye. *Toward Guiding Simulation Experiments*. PhD thesis, University of Rostock, Germany, 2014.

[117] Stefan Leye, Roland Ewald, and Adelinde M. Uhrmacher. Composing Problem Solvers for Simulation Experimentation: A Case Study on Steady State Estimation. *PLoS ONE*, 9(4):e91948, 2014.

[118] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical Hardness Models: Methodology and a Case Study on Combinatorial Auctions. *Journal of the ACM*, 56(4):22:1–22:52, 2009.

[119] Hong Li and Linda Petzold. Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems. Technical report, Department of Computer Science, University of California: Santa Barbara, 2006.

[120] Karl Lieberherr. *Adaptive Object-Oriented Software The Demeter Method*. 1996.

[121] J. Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information Theory*, 37(1):145–151, 1991.

[122] Martin Lippert and Cristina Videira Lopes. A Study on Exception Detection and Handling Using Aspect-oriented Programming. In *Proceedings of the 22Nd International Conference on Software Engineering*, pages 418–427, 2000.

[123] Martin Luboschik, Stefan Rybacki, Roland Ewald, Benjamin Schwarze, Heidrun Schumann, and Adelinde M Uhrmacher. Interactive Visual Exploration of Simulator Accuracy: A Case Study for Stochastic Simulation Algorithms. In *Proceedings of the 44th Winter Simulation Conference (WSC'12)*, pages 1–12, 2012.

[124] W. J. Conover M. D. McKay, R. J. Beckman. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, 21(2):239–245, 1979.

[125] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 147–155, 1987.

[126] Paolo Marrone. JOONE: The Complete Guide, 2004.

[127] Nazzareno Marziale, Francesco Nobilia, Alessandro Pellegrini, and Francesco Quaglia. Granular Time Warp Objects. In *Proceedings of the ACM Conference on Principles of Advanced Discrete Simulation (PADS)*, pages 57–68, 2016.

[128] Sean Mauch and Mark Stalzer. Efficient Formulations for Exact Stochastic Simulation of Chemical Systems. *IEEE/ACM Transactions on Computing Biology and Bioinformatics*, 8(1):27–35, 2011.

[129] Carsten Maus. *Toward Accessible Multilevel Modeling in Systems Biology - A Rule-based Language Concept*. PhD thesis, University of Rostock, Germany, 2012.

[130] Carsten Maus, Stefan Rybacki, and Adelinde M Uhrmacher. Rule-based multi-level modeling of cell biological systems. *BMC Systems Biology*, 5(166), 2011.

[131] Orianne Mazemondet, Rayk Hubner, Jana Frahm, Dirk Koczan, Benjamin M Bader, Dieter G Weiss, Adelinde M Uhrmacher, Moritz J Frech, Arndt Rolfs, and Jiankai Luo. Quantitative and kinetic profile of Wnt/$\beta$-catenin signaling components during human neural progenitor cell differentiation. *Cellular & Molecular Biology Letters*, 16(4):515–538, 2011.

[132] Orianne Mazemondet, Mathias John, Stefan Leye, Arndt Rolfs, and Adelinde M. Uhrmacher. Elucidating the Sources of $\beta$-Catenin Dynamics in Human Neural Progenitor Cells. *PLoS ONE*, 7(8), 2012.

[133] James M. McCollum, Gregory D. Peterson, Chris D. Cox, Michael L. Simpson, and Nagiza F. Samatova. The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior. *Computational Biology and Chemistry*, 30(1):39–49, 2006.

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

[135] Philip K McKinley, Seyed M Sadjadi, Eric P Kasten, and Betty H C Cheng. Composing Adaptive Software. *Computer*, 37(7):56–64, July 2004.

[136] Steffen Mentel. *Konfiguration dynamischer Zustandsräume für adaptive Simulationsalgorithmen*. Bachelor's thesis, University of Rostock, 2014.

[137] Sina Meraji, Carl Tropper, and Wei Zang. A Multi-State Q-learning Approach for the Dynamic Load Balancingof Time Warp. In *Proceedings of the 24th Workshop on Principles of Advanced and Distributed Simulation (PADS'10)*, pages 1–8, May 2010.

[138] Peter Meso and Radhika Jain. Agile Software Development: Adaptive Systems Principles and Best Practices. *Information Systems Management*, 23(3):19–30, 2006.

[139] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten. Does Aspect-oriented Programming Work? *Communications of the ACM*, 44(10):75–77, 2001.

[140] Mohammad Reza Nami and Mohsen Sharifi. A Survey of Autonomic Computing Systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic and Autonomous Systems*, pages 101–110, 2007.

[141] Egbert H. Van Nes and Marten Scheffer. A strategy to improve the contribution of complex simulation models to ecological theory. *Ecological Modelling*, 185(2–4):153–164, 2005.

[142] David M Nicol and Paul F Reynolds Jr. Optimal Dynamic Remapping of Data Parallel Computations. *IEEE Transactions on Computers*, 39(2):206–219, February 1990.

[143] Jürgen Nievergelt, Hans Hinterberger, and Kenneth C Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[144] Denis Noble. *The Music of Life: Biology Beyond Genes*. Oxford University Press, 2006.

[145] Scott Oaks. *Java Performance: The Definitive Guide*. O'Reilly Media, 2014.

[146] T. Oguara, D. Chen, G. Theodoropoulos, B. Logan, and M. Lees. An Adaptive Load Management Mechanism for Distributed Simulation of Multi-agent Systems. In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)*, pages 179–186, 2005.

[147] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[148] Jürgen Pahle. Biochemical simulations: stochastic, approximate stochastic and hybrid approaches. *Briefings in Bioinformatics*, 10(1):53–64, 2009.

[149] Avinash C Palaniswamy and Philip A Wilsey. Adaptive bounded time windows in an optimistically synchronized simulator. In *Proceedings of the 3rd Great Lakes Symposium on Design Automation of High Performance VLSI Systems (VLSI'93)*, pages 114–118, March 1993.

[150] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[151] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[152] Ricardo Paxson and Kristen Zannella. Studying the World's Most Complex Dynamic Systems. *The MathWorks News&Notes*, 2007.

[153] Danhua Peng, Tom Warnke, Fiete Haack, and Adelinde M. Uhrmacher. Reusing simulation experiment specifications in developing models by successive composition — a case study of the Wnt/$\beta$-catenin signaling pathway. 2017. In Press.

[154] L. Felipe Perrone, Cristopher S. Main, and Brian. C. Ward. SAFE: Simulation Automation Framework for Experiments. In *Proceedings of the Winter Simulation Conference (WSC)*, pages 1–12, 2012.

[155] Kalyan S. Perumalla. Parallel and Distributed Simulation: Traditional Techniques and Recent Advances. In *Proceedings of the 38th Winter Simulation Conference*, pages 84–95, 2006.

[156] Patrick Peschlow, Tobias Honecker, and Peter Martini. A Flexible Dynamic Partitioning Algorithm for Optimistic Distributed Simulation. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, pages 219–228, 2007.

[157] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Universität Bonn, 1962.

[158] Linda Petzold. Automatic Selection of Methods for Solving Stiff and Nonstiff Systems of Ordinary Differential Equations. *SIAM Journal on Scientific and Statistical Computing*, 4(1):136–148, 1983.

[159] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, 3 edition, 2007.

[160] Jacek Puchałka and Andrzej M. Kierzek. Bridging the gap between stochastic and deterministic regimes in the kinetic simulations of the biochemical reaction networks. *Biophysical Journal*, 86(3):1357–1372, 2004.

[161] Francesco Quaglia. A scaled version of the elastic time algorithm. In *Proceedings of the 15th Workshop on Parallel and Distributed Simulation (PADS)*, pages 157–164, 2001.

[162] Muruhan Rathinam, Linda R. Petzold, Yang Cao, and Daniel T. Gillespie. Stiffness in stochastic chemically reacting systems: The implicit tau-leaping method. *The Journal of Chemical Physics*, 119(24):12784–12794, 2003.

[163] P. L. Reiher, F. Wieland, and D. Jefferson. Limitation of Optimism in the Time Warp Operating System. In *Proceedings of the 21st Winter Simulation Conference*, pages 765–770, 1989.

[164] Oliver Reinhardt. *Bayessche Changepoint-Detection zur Unterstützung adaptiver Simulationsalgorithmen.* Bachelor's thesis, University of Rostock, 2014.

[165] Stuart I. Reynolds. Decision boundary partitioning: Variable resolution model-free reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning (ICML'00)*, pages 783–790, 2000.

[166] John R Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[167] Stefan Rybacki, Fiete Haack, Karsten Wolf, and Adelinde M. Uhrmacher. Developing Simulation Models - from Conceptual to Executable Model and Back - an Artifact-based Workflow Approach. In *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques*, pages 21–30, 2014.

[168] S. M. Sadjadi. A Survey of Adaptive Middleware. Technical report, Software Engineering and Network Systems Laboratory, Department of Computer Science and EngineeringMichigan State University.

[169] S. Masoud Sadjadi and Fernando Trigoso. TRAP.NET: A REALIZATION OF TRANSPARENT SHAPING IN .NET. *International Journal of Software Engineering and Knowledge Engineering*, 19(04):507–528, 2009.

[170] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14:1–14:42, 2009.

[171] Howard Salis and Yiannis Kaznessis. Accurate hybrid stochastic simulation of a system of coupled chemical or biochemical reactions. *The Journal of Chemical Physics*, 122(5), 2005.

[172] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

[173] Werner Sandmann. Streamlined formulation of adaptive explicit-implicit tau-leaping with automatic tau selection. In *Proceedings of the 41st Winter Simulation Conference (WSC)*, pages 1104–1112, 2009.

[174] Johannes Schützel, Holger Meyer, and Adelinde M. Uhrmacher. A Stream-based Architecture for the Management and On-line Analysis of Unbounded Amounts of Simulation Data. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 83–94, 2014.

[175] R. Sherer, A. Gupta, and M. Hybinette. Adaptive message clustering for distributed agent-based systems. In *IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 1–6, 2011.

[176] David J Sheskin. *Handbook of parametric and nonparametric statistical procedures*. Chapman & Hall/CRC, 4th edition, 2007.

[177] Dilma M. da Silva and Fabio Kon. Adaptive software systems. *Journal of the Brazilian Computer Society*, 10:3–4, 2004.

[178] Christopher Simpkins, Sooraj Bhat, Charles Isbell, Jr., and Michael Mateas. Towards Adaptive Programming: Integrating Reinforcement Learning into a Programming Language. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 603–614, 2008.

[179] Michael W Sneddon, James R Faeder, and Thierry Emonet. Efficient modeling, simulation and coarse-graining of biological complexity with NFsim. *Nature Methods*, 8(2):177–183, 2011.

[180] Robert R Sokal and F James Rohlf. Biometry: the principals and practice of statistics in biological research. *W.H. Freeman and Company, New York*, 1995.

[181] Lisa M. Sokol, Jon B. Weissman, and Paula A. Mutchler. MTW: An Empirical Performance Study. In *Proceedings of the 23rd Winter Simulation Conference*, pages 557–563, 1991.

[182] H. M. Soliman and A. S. Elmaghraby. An Efficient Clustered Adaptive-Risk Technique for Distributed Simulation. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, pages 383–391, 1996.

[183] Sudhir Srinivasan and Paul F. Reynolds, Jr. Elastic Time. *ACM Transactions of Modeling and Computer Simulation*, 8(2):103–139, 1998.

[184] Remo Suppi, Fernando Cores, and Emilio Luque. Improving Optimistic PDES in PVM Environments. In *Recent Advances in Parallel Virtual Machine and*

*Message Passing Interface: Proceedings of the 7th European PVM/MPI Users' Group Meeting Balatonfüred*, pages 304–312, 2000.

[185] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[186] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous Nearest Neighbor Search. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 287–298, 2002.

[187] Andre L.C. Tavares and Marco Tulio Valente. A Gentle Introduction to OSGi. *ACM SIGSOFT Software Engineering Notes*, 33(5):8:1–8:5, 2008.

[188] V.E. Taylor, B.K. Holmer, E.J. Schwabe, and M.R. Hribar. Balancing load versus decreasing communication: exploring the tradeoffs. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, volume 1, pages 585–593, 1996.

[189] Thanh, Vo Hong and Priami, Corrado and Zunino, Roberto. Efficient rejection-based simulation of biochemical reactions with stochastic noise and delays. *The Journal of Chemical Physics*, 141(13), 2014.

[190] Adelinde M. Uhrmacher. Dynamic Structures in Modeling and Simulation: A Reflective Approach. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11(2):206–232, 2001.

[191] Adelinde M. Uhrmacher, Jan Himmelspach, and Roland Ewald. Effective and Efficient Modeling and Simulation with DEVS Variants. *Discrete-Event Modeling and Simulation: Theory and Applications*, 2011.

[192] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[193] Joannès Vermorel and Mehryar Mohri. Multi-armed Bandit Algorithms and Empirical Evaluation. In *Proceedings of the 16th European conference on Machine Learning (ECML'05)*, pages 437–448, 2005.

[194] Ricardo Vilalta and Youssef Drissi. A Perspective View and Survey of Meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

[195] Tom Warnke, Tobias Helms, and Adelinde M. Uhrmacher. Syntax and Semantics of a Multi-Level Modeling Language. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 133–144, 2015.

[196] Christopher Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, England, 1989.

[197] Andreas Weidemann, Stefan Richter, Matthias Stein, Sven Sahle, Ralph Gauges, Razif Gabdoulline, Irina Surovtsova, Nils Semmelrock, Bruno Besson, Isabel Rojas, Rebecca Wade, and Ursula Kummer. SYCAMORE – a systems biology computational analysis and modeling research environment. *Bioinformatics*, 24(12):1463–1464, 2008.

[198] Pia Wilsdorf. *Approximation diskreter Ereignisse bei hybrid ausgeführten ML-Rules Simulationen*. Bachelor's thesis, University of Rostock, 2016.

[199] James R. Wilson and A. Alan B. Pritsker. A survey of research on the simulation startup problem. *SIMULATION*, 31(2):55–58, 1978.

[200] Robert C Wilson, Matthew R Nassar, and Joshua I Gold. Bayesian Online Learning of the Hazard Rate in Change-Point Problems. *Neural computation*, 22(9):2452–2476, 2010.

[201] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3 edition, 1 2011.

[202] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically Configuring Algorithms for Portfolio-Based Selection. 2010.

[203] Richard J. Youle and Alexander M. van der Bliek. Mitochondrial Fission, Fusion, and Stress. *Science*, 337(6098):1062–1065, 2012.

[204] Jakob Zabel. *Statistical Testing of Component-based Stochastic Simulation Algorithms*. Master's thesis, University of Rostock, 2014.

[205] Bernard P Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. Academic Press, Inc., 2nd edition, 2000.

# Appendix A

# ML-Rules Models

All models are given in the current syntax of ML-Rules. For more information, see the ML-Rules repository at `https://git.informatik.uni-rostock.de/mosi/mlrules2`.

## A.1 Cell Cycle Model

```
1
2  // PARAMETERS
3  ctot:10; dtot:1000; k1:0.015*dtot; k2:200; k3:180; k3prime:0.018;
4  k4:4.5; k5:0.6; k6:1.0; k7:1e6; k8:k7; k9:k7; t7:250; t8:70; t9:20;
5  td:116;
6
7  // ENTITY DEFINITIONS
8  C(num,string)[]; Y(); Yp(); D(); Mi(); Ma();
9
10 // INITIAL SOLUTION
11 >>INIT [
12    ctot C(1.0, 'G1')[(dtot-1) D() + 1 Ma()]
13 ];
14
15 // RULE SCHEMAS
16
17 // cyclin synthesis
18 C(v, p)[sol?]:c -> C(v, p)[Y() + sol?]
19    @ k1*#c;
20
21 // formation of inactive MPF complex
22 C(v, p)[Y():y + D():d + sol?]:c -> C(v, p)[Mi() + sol?]
23    @ k2*#y*#d*#c;
24
```

```
25 // activation of MPF complex
26 Mi():i + Ma():a -> 2Ma()
27   @ (k3prime+(k3*((#a/dtot)^(2))))*#i;
28
29 // breakage of activated MPF complex
30 C(v, p)[Ma():a + sol?]:c -> C(v, p)[Yp() + D() + sol?]
31   @ if (#a>1) then (k4/v)*#a*#c else 0;
32
33 // cyclin degradation
34 Yp():y ->
35   @ k5*#y;
36
37 // cell growth
38 C(v, p)[sol?]:c -> C(v+(1/td), p)[sol?]
39   @ if (p=='G1') || (p=='SG2') then k6*#c else 0;
40
41 // cell cycle transition from G1->S/G2
42 C(v, 'G1')[Mi():i + sol?]:c -> C(v, 'SG2')[Mi() + sol?]
43   @ if (#i>t7) then k7*#c else 0;
44
45 // cell cycle transition from S/G2->M
46 C(v, 'SG2')[Ma():a + sol?]:c -> C(v, 'M')[Ma() + sol?]
47   @ if (#a>t8) then k8*#c else 0;
48
49 // cell division (transition from M->G1)
50 C(v, 'M')[Ma():a + sol?]:c -> C(v/2, 'G1')[Ma() + sol?]
51   @ if (#a<t9) then k9*#c else 0;
```

## A.2 Endocytosis Model

```
1  // PARAMETERS
2  kendo:0.001; kfuse:0.002; krecrR7:0.001; krecrR5:0.001; tpH:3.0;
3  kextrR7:0.001; kextrR5:0.01; krecycle:1;
4
5  // SPECIES DEFINITIONS
6  Particle(); Cell()[]; Vesicle()[]; Endosome(num,string,num)[];
7  Lysosome()[]; Rab5(); Rab7();
8
9  // INITIAL SOLUTION
10 >>INIT[
11   200 Particle() +
12   1 Cell[
13     1 Lysosome() + 5000 Rab5() + 5000 Rab7()
14   ]
```

```
15  ];
16
17  // RULE SCHEMATA
18
19  // vesicle budding (process of endocytosis)
20  Particle:p + Cell[sol?] -> Cell[Vesicle[Particle] + sol?]
21    @ kendo*#p;
22
23  // vesicle/vesicle fusion
24  Vesicle[sol1?] + Vesicle[sol2?]
25    -> Endosome(2.0,'early',8.0)[sol1? + sol2?]
26    @ kfuse;
27
28  // vesicle/endosome fusion (volume and pH are adjusted)
29  Vesicle[sol1?] + Endosome(vol,'early',pH)[sol2?]
30    -> Endosome(vol+1.0,'early',((pH*vol)+8)/(vol+1.0))[sol1? + sol2?]
31    @ kfuse;
32
33  // endosome/endosome fusion (volume and pH are adjusted)
34  Endosome(vol1,state,pH1)[sol1?] + Endosome(vol2,state,pH2)[sol2?]
35    -> Endosome(vol1+vol2,state,((pH1*vol1)+(pH2*vol2))/(vol1+vol2))[
36          sol1? + sol2?
37        ]
38    @ kfuse;
39
40  // state change of endosomes from early to late
41  Endosome(volume,'early',pH)[Rab5:r5 + Rab7:r7 + sol?]
42    -> Endosome(volume,'late',pH)[Rab5 + Rab7 + sol?]
43    @ if (#r7>(2*#r5)) then 1 else 0;
44
45  // endosome/lysosome fusion
46  Endosome(volume,'late',pH)[sol1?] + Lysosome()[sol2?]
47   -> Lysosome()[sol1? + sol2?]
48    @ kfuse;
49
50  // pH decrease within endosomes
51  Endosome(volume,state,pH)[sol?]
52    -> Endosome(volume,state,pH-(log(pH)/1000))[sol?]
53    @ 10;
54
55  // Rab7 recruitment
56  Endosome(volume,state,pH)[sol?] + Rab7:r7
57    -> Endosome(volume,state,pH)[Rab7 + sol?]
58    @ #r7*krecrR7;
59
60  // Rab5 recruitment (basal and with positive feedback)
```

```
61 Endosome(volume,state,pH)[sol?] + Rab5:r5
62   -> Endosome(volume,state,pH)[Rab5 + sol?]
63   @ #r5*krecrR5;
64 Endosome(volume,state,pH)[Rab5:r5 + sol?] + Rab5:r52
65   -> Endosome(volume,state,pH)[2 Rab5 + sol?]
66   @ if (pH>tpH) then #r52*(#r5/100)*krecrR5 else 0;
67
68 // Rab5 extraction
69 Endosome(volume,state,pH)[Rab5:r5 + sol?]
70   -> Endosome(volume,state,pH)[sol?] + Rab5
71   @ #r5*kextrR5;
72
73 // Rab7 extraction
74 Endosome(volume,state,pH)[Rab7:r7 + sol?]
75   -> Endosome(volume,state,pH)[sol?] + Rab7
76   @ #r7*kextrR7;
77
78 // Rab5 and Rab7 recycling from lysosome
79 Lysosome()[Rab5:r5 + sol?] -> Lysosome()[sol?] + Rab5 @ #r5*krecycle;
80 Lysosome()[Rab7:r7 + sol?] -> Lysosome()[sol?] + Rab7 @ #r7*krecycle;
81
82 // Particle degradation
83 Lysosome()[Particle:p + sol?] -> Lysosome()[sol?] @ #p*0.01;
```

## A.3 Wnt/$\beta$-catenin Model

```
1  // initial species counts
2  nbetacyt: 12989; nbetanuc: 5282; nAxin: 252; nAxinP: 219; nWnt: 1000;
3  nCells: 1;
4
5  // reaction rate coefficients
6  kbetasyn: 600; kWdeg: 0.27; kApA_act: 20; kApA: 0.03; kAAp: 0.03;
7  kApdeg: 4.48E-3; kAdeg: 4.48E-3; kbetadeg_act: 2.1E-4;
8  kbetadeg: 1.13E-4; kbetain: 0.0549; kbetaout: 0.135; kAsyn: 4E-4;
9
10 // species definitions (number of attributes)
11 Cell(string,num)[]; // cell cycle phase / cytosolic compartment volume
12 Nuc(num)[];  // compartment volume
13 Wnt();
14 Axin(string); // phosphorylation state
15 Bcat();
16
17 // initial solution
18 >>INIT[
```

```
19            (nWnt) Wnt +
20            nCells Cell('G1',1)[
21              (nbetacyt) Bcat +
22              nAxin Axin('u') +
23              nAxinP Axin('p') +
24              Nuc(1)[(nbetanuc) Bcat]
25            ]
26  ];
27
28
29  // (1) Wnt degradation
30  Wnt:w -> @ kWdeg*#w;
31
32  // (2) activated AxinP dephosphorylation
33  // (prepared for dynamic compartment volume)
34  Wnt:w + Cell(phase,vol)[Axin('p'):a + s?]
35    -> Wnt + Cell(phase,vol)[Axin('u') + s?]
36    @ ((kApA_act*#w*#a)/vol);
37
38  // (3) basal AxinP dephosphorylation
39  Axin('p'):a -> Axin('u') @ kApA*#a;
40
41  // (4) Axin phosphorylation
42  Axin('u'):a -> Axin('p') @ kAAp*#a;
43
44  // (5) AxinP degradation
45  Axin('p'):a -> @ kApdeg*#a;
46
47  // (6) Axin degradation
48  Axin('u'):a -> @ kAdeg*#a;
49
50  // (7) activated beta-catenin degradation
51  // (prepared for dynamic compartment volume)
52  Cell(phase,vol)[Axin('p'):a + Bcat:b + s?]
53    -> Cell(phase,vol)[Axin('p') + s?]
54    @((kbetadeg_act*#a*#b)/vol);
55
56  // (8) beta-catenin synthesis
57  Cell(phase,vol)[s?] -> Cell(phase,vol)[Bcat + s?] @ kbetasyn;
58
59  // (9) basal beta-catenin degradation
60  Bcat:b -> @ kbetadeg*#b;
61
62  // (10) beta-catenin shuttling into the nucleus
63  Bcat:b + Nuc(vol)[s?] -> Nuc(vol)[Bcat + s?] @ kbetain*#b;
64
```

```
65 // (11) beta-catenin shuttling out of the nucleus
66 Nuc(vol)[Bcat:b + s?] -> Bcat + Nuc(vol)[s?] @ kbetaout*#b;
67
68 // (12) Axin synthesis
69 Nuc(vol)[Bcat:b + s?] -> Nuc(vol)[Bcat + s?] + Axin('u') @ kAsyn*#b;
```

## A.4 Simplified Lipid Raft Model

```
1  // initial species counts
2  nR: 1000;  // per sub
3  nLR: 10;  // per volume
4
5  // reaction rate coefficients
6  kLRsyn: 1.0; kLRdeg: 0.1; kRsyn: 50; kRin: 1; kRout: 1;
7  volume: 2000; // volume per sub volume
8  dd:1; // diffusion coefficient
9  rho:0.3; // raft fluidity
10
11 R(num,num);    //Diffusion, isInRaft
12 LR(num,num)[];  // radius, fluidity
13 SubVol(num,num,num)[]; // x and y coordinates and volume
14
15 // initial solution
16 >>INIT[
17         SubVol(1,1,volume)[nLR LR(4, rho) + nR R(dd,0)]
18 ];
19
20 // (1) Raft degradation
21 LR(vol, p):l -> @ #l*kLRdeg;
22
23 // (2) Raft synthesis
24 SubVol(x,y,v)[s?] -> SubVol(x,y,v)[LR(4, rho) + s?] @kLRsyn;
25
26 // (3) Receptor synthesis
27 SubVol(x,y,v)[s?] -> SubVol(x,y,v)[R(dd, 0) + s?] @kRsyn;
28
29 // (4) Receptor diffusion into LR
30 LR(vol, p)[sol?] + R(d, 0):r -> LR(vol, p)[R(d*p, 1) + sol?]
31   @ (kRin*(4*3.14*d*vol)*(#r/(volume-(3.14*vol*vol))));
32
33 // (5) Receptor diffusion out of LR
34 LR(vol, p)[R(d, 1):r + sol?] -> LR(vol, p)[sol?] + R(d/p, 0)
35   @ (kRout*(4*3.14*d*vol)*(#r/(3.14*vol*vol)));
```

## A.5 Dictyostelium Discoideum Model

```
//Dictyostelium aggregation model in discrete space

// PARAMETERS
scale:1;
// default: one cell per grid position
xmax:2; ymax:2; kd_dicty:2/1000; kd_camp:2.4e6/(1000*1000);
nA:6.023e23; v:3.6720e-14;
k1:2.0; k2:0.9/nA/(v*scale)/1e-6; k3:2.5; k4:1.5; k5:0.6;
k6:0.8/nA/(v*scale)/1e-6; k7:1.0*nA*(v*scale)*1e-6;
k8:1.3/nA/(v*scale)/1e-6; k9:0.3; k10:0.8/nA/(v*scale)/1e-6;
k11:0.7; k12:4.9; k13:23.0; k14:4.5;
init_cAMPe:1100*scale;
init_cAMPi:4100*scale;
init_ACA:7300*scale;
init_PKA:7100*scale;
init_ERK2:2500*scale;
init_RegA:3000*scale;
init_CAR1:6000*scale;

first :: num -> sol;
first 0 = [];
first x = second(xmax,x) + first(x-1);

second :: num -> num -> sol;
second 0 y = [];
second x y = (init_cAMPe) CAMPe(x,y) + 1 CELL(x,y)[
                                          (init_cAMPi) CAMPi
                                          + (init_ACA) ACA
                                          + (init_PKA) PKA
                                          + (init_ERK2) ERK2
                                          + (init_RegA) RegA
                                          + (init_CAR1) CAR1
                                        ]
              + second(x-1,y);

// SPECIES DEFINITIONS
System()[]; CELL(num,num)[]; CAMPe(num,num); CAMPi(); ACA();
PKA(); ERK2(); RegA(); CAR1();

// INITIAL SOLUTION
>>INIT[
System[first(ymax)]
];

```

```
45 // REACTION RULES
46
47 // intra-cellular dynamics
48 CAR1:c -> ACA + CAR1 @ k1*#c;
49 ACA:a + PKA:p -> PKA @ k2*#a*#p;
50 CAMPi:a -> PKA + CAMPi @ k3*#a;
51 PKA:p -> @ k4*#p;
52 CAR1:c -> ERK2 + CAR1 @ k5*#c;
53 PKA:p + ERK2:e -> PKA @ k6*#p*#e;
54 CELL(x,y)[s?]:c -> CELL(x,y)[RegA + s?] @ k7*#c;
55 ERK2:e + RegA:r -> ERK2 @ k8*#e*#r;
56 ACA:a -> CAMPi + ACA @ k9*#a;
57 RegA:r + CAMPi:a -> RegA @ k10*#r*#a;
58 CELL(x,y)[ACA:a + s?] -> CAMPe(x,y) + CELL(x,y)[ACA + s?] @ k11*#a;
59 CAMPe(x,y):a -> @ k12*#a;
60 System[CAMPe(x,y):a + CELL(x,y)[c?] + r?]
61   -> System[CELL(x,y)[CAR1 + c?] + CAMPe(x,y) + r?]
62   @ k13*#a/(1 + countTwoAtts(r?,'Cell',x,y));
63 CAR1:c -> @ k14*#c;
64
65 // movement of cell to adjacent position depending on external cAMP
     amount
66 CELL(x1,y1)[s?] + CAMPe(x1,y1):a1 + CAMPe(x2,y2):a2
67   -> CELL(x2,y2)[s?] + CAMPe(x1,y1) + CAMPe(x2,y2)
68   @ if ((#a2>#a1) && (#a1 > 0) && ((x1!=x2) && (y1!=y2)) &&
     ((x1-x2<=1)&&(x1-x2>=-1)) && ((y1-y2<=1)&&(y1-y2>=-1))) then
     kd_dicty*(#a2*(1/#a1)) else 0;
69
70 // cAMP diffusion
71 CAMPe(x,y):a -> CAMPe(x,y+1)
72   @ if (y<ymax) then kd_camp*#a else 0;
73 CAMPe(x,y):a -> CAMPe(x+1,y+1)
74   @ if (x<xmax) && (y<ymax) then kd_camp*#a else 0;
75 CAMPe(x,y):a -> CAMPe(x+1,y)
76   @ if (x<xmax) then kd_camp*#a else 0;
77 CAMPe(x,y):a -> CAMPe(x+1,y-1)
78   @ if (x<xmax) && (y>1) then kd_camp*#a else 0;
79 CAMPe(x,y):a -> CAMPe(x,y-1)
80   @ if (y>1) then kd_camp*#a else 0;
81 CAMPe(x,y):a -> CAMPe(x-1,y-1)
82   @ if (x>1) && (y>1) then kd_camp*#a else 0;
83 CAMPe(x,y):a -> CAMPe(x-1,y)
84   @ if (x>1) then kd_camp*#a else 0;
85 CAMPe(x,y):a -> CAMPe(x-1,y+1)
86   @ if (x>1) && (y<ymax) then kd_camp*#a else 0;
```

# Thesis Statements

1. Many existing effective methods adapting simulators emphasize that simulator adaptations at runtime is a valuable approach to improve the efficiency of simulation runs.

2. Component-based simulation systems like JAMES II offer a great flexibility, but they also come along with complex selection challenges that should be solved automatically.

3. A generic adaptive simulator has been developed and integrated into JAMES II. It encapsulates available simulators applicable to a specific problem, employs reinforcement learning to explore and exploit the performance of these simulators, and exchanges the currently used encapsulated simulator as needed.

4. As the developed adaptive simulator uses the encapsulated simulators to calculate the state transitions of a model, it is not restricted to any modeling language, but it can be applied to all modeling approaches available in JAMES II.

5. Using changepoint detection considering the event throughput to trigger adaptations for the developed adaptive simulator showed to be robust and effective.

6. Dynamic state space generalization algorithms can significantly improve the learning efficiency of the developed adaptive simulator. The decision boundary partitioning algorithm can perform better than the adaptive vector quantization, but it is less robust.

7. ML-Rules is an expressive modeling language inducing various computational challenges making it a suitable language to develop component-based, tailored and approximate simulators.

8. A subset of ML-Rules models with a fixed reaction network can be simulated more efficiently by using a tailored simulator only applicable to such models.

9. ML-Rules models focusing on species bonds tend to consist of rules fulfilling the rigidity property. Exploiting this property by tailored simulators allows for a better runtime performance.

10. Approximate ML-Rules simulators should partition reactions by their influence on the reaction network. Only reactions not changing the reaction network should be approximated. $\tau$-leaping as well as a hybrid simulator showed to be beneficial compared to the exact ML-Rules simulator.

# Erklärung

Ich, Tobias Helms, erkläre, dass ich die vorliegende Dissertationsschrift mit dem Thema: "Simulator Adaptation at Runtime for Component-based Simulation Software" selbständig, ohne die (unzulässige) Hilfe Dritter und nur unter der Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.