Model-Based Testing for Self-Organization Mechanisms

Dissertation

zur Erlangung des akademischen Grades
doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Benedikt Eberhardinger

an der

Fakultät für Angewandte Informatik

am

14. September 2018

Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Drittgutachter:	Prof. Dr. Franz Wotawa (Technische Universität Graz, Österreich)

Tag der mündlichen Prüfung: 3. Dezember 2018

For my beloved wife and daughter.

Acknowledgements

This thesis would not have been possible without the tremendous support from different sides. I would like to take this opportunity to thank all the people who supported me during the time from the very beginning until the completion of this thesis.

First, I would like to express my appreciation and thanks to my advisor Professor Dr. Wolfgang Reif who granted me his confidence and gave me the opportunity to carry out my research at his institute. He gave me the freedom to go into new directions but still kept me from drifting off. I would like to thank him for the many advises, the numerous creative discussions that all have contributed to this thesis.

Further, I would like to thank Professor Dr. Bernhard Bauer and Professor Dr. Franz Wotawa for declaring themself ready to comply with being my secondary and third assessors for this thesis.

A lot of the results that formed this thesis have been a result of intense cooperation with my great colleagues and dear friends from the Institute for Software & Systems Engineering. I would like to thank all of them for their countless comments at our seminars and for all the fruitful collaborations and discussions. Special thanks are dedicated to my colleges Dr. Hella Ponsar, Dr. Axel Habermaier, André Reichstaller, Professor Dr. Alexander Knapp, Dr. Alexander Schiendorfer, Professor Dr. Jan-Philipp Steghöfer, Dr. Gerrit Anders, and Dr. Florian Siefert for the successful and exciting collaboration in our joint projects. Further, I like to thank Dominik Klumpp and Gerald Siegert for their collaboration in the context of their bachelor thesis, master thesis, and their work as research assistants at our institute, whereby they supported my research in many regards.

For the intensive and mindful feedback on this thesis, I gratefully thank Dr. Hella Ponsar, Dr. Andreas Angerer, Dr. Axel Habermaier, Dr. Alexander Schiendorfer, and Professor Dr. Alexander Knapp,. With their comments and advice, I was able to put the finishing touches on this thesis.

On a final note, I like to thank my family and friends that supported me during this thesis and beyond. They always took off the load from me when necessary and helped me by believing in my success. Foremost, I like to thank my wife Clarissa who always lend me her support, kept up my motivation, and sustained me in situations where it was most needed. Thank you for your patience and confidence in me. Without your support, I would not have been able to complete this thesis.

Benedikt Eberhardinger

Abstract

A complex system is a system composed of many different (often heterogeneous) components that interact with each other. These complex systems become more ubiquitous, e.g., in cloud computing, the internet of things, or industry 4.0. Self-Organization (SO) is offering new flexibility that can organize this complexity: Self-Organization enables a restructuring of a system and its components at run time in order to conform to the system's dynamic and ever-changing environment and to fulfill its goals without human intervention. The complexity of the system arises either from the lack of knowledge at design time or the hardly predictable operational conditions of the systems at run time. However, the newly gained flexibility through the introduction of SO comes at a price: the design and implementation of SO mechanisms are a demanding engineering task. Whereas there is much research in the design and implementation of SO mechanisms, there is a lack of adequate testing techniques. This thesis narrows the gap by providing a Model-Based Testing (MBT) approach for SO mechanisms.

Six significant contributions constitute the approach: Revealing failures—the intent of testing—demands a definition of the correct and the incorrect behavior of the investigated System under Test (SuT). In general, the correct behavior of an SuT is specified, and any deviation is not intended. Judging about failures is difficult in SO systems which might recover from a situation where the specification is only temporarily not fulfilled. This thesis thus offers *a new notion of failure for SO mechanisms* that copes with the specific behavior of SO mechanisms. Self-Organization mechanisms are highly interwoven with the controlled system and its environment. This characteristic makes it difficult to control and observe the during testing. With the architectural pattern of the Corridor Enforcing Infrastructure (CEI), this thesis offers *a generic approach for making SO mechanisms testable*. Further, we show that a broad class of SO systems is testable within this approach. Moreover, the concepts of the CEI allow for *systematically decomposed and assembled SO mechanisms for isolated and integrated testing*.

The resulting test architecture, presented in this thesis, includes all these contributions and offers a complete test scaffold. Within this scaffold, an MBT approach for SO mechanisms is provided. This approach is *suited for SO mechanisms by extending MBT with a so-called models at run time approach that allows for closing the feedback loop of MBT*. Thus, it is possible to use the feedback of the Self-Organization Mechanism under Test (SOuT) during execution in the test model. This knowledge is exploited in a fault-based and a probabilistic model for test case generation. Within this model, we can apply a suitable *search-based test generation approach for efficient testing of SO mechanisms*. Thus, we can exploit unique characteristics of SO mechanisms to speed up failure detection by up to 500 times. Testing the functional requirements of an SO mechanisms is the focus of this thesis. However, we are also able to show the ability of the presented MBT approach to *investigate the performance of SO mechanisms*.

In order to demonstrate the capabilities of the presented approach, it is evaluated on five different case studies incorporating different kinds of SO mechanisms that originate from different companies, institutions, and researchers.

Contents

1 Overview and Motivation **1**
- 1.1 Key Challenges in Testing Self-Organization Mechanisms 2
- 1.2 Model-Based Testing for Self-Organization Mechanisms 5
- 1.3 Main Contributions and Thesis Outline 7

2 Case Studies **11**
- 2.1 Case Studies with Continuous Self-Organization Mechanisms 12
  - 2.1.1 Decentralized Power Management 12
  - 2.1.2 Self-Adaptive Apache Hadoop Manager 15
- 2.2 Case Studies with Discrete Self-Organization Mechanisms 17
  - 2.2.1 Self-Organization Production Cell 17
  - 2.2.2 Self-Organized Personalized Medicine Pill Production System . 20
  - 2.2.3 Self-Adaptive Webservice System: ZNN.com 22

3 Specification of Functional Behavior of Self-Organization Mechanisms and Derivation of an Automated Test Oracle **25**
- 3.1 Related Work 27
  - 3.1.1 Specification of Self-Organizing, Adaptive System (SOAS) . . . 27
  - 3.1.2 Deriving Automated Test Oracles 28
  - 3.1.3 Runtime Verification 29
- 3.2 The Corridor of Correct Behavior—Specification of Self-Organizing Behavior 30
  - 3.2.1 The Restore Invariant Approach—Descirbing the Corridor of Correct Behavior 30
  - 3.2.2 Application of the Restore Invariant Approach (RIA) to Software Testing 31
- 3.3 Goal-oriented Modeling of Functional Behavior with KAOS 32
  - 3.3.1 The KAOS Methodology 32
  - 3.3.2 RELAX Goals for Introducing SO as Adaptation 33
- 3.4 Deriving the Test Oracle from the KAOS Model 35
  - 3.4.1 Process for Generating an Automated Test Oracle 36
  - 3.4.2 Implementation of Transforming Requirement and Constraints to a Monitor Model 37
  - 3.4.3 Implementation of the Transformation for the Monitor Model to an Oracle 39

4 A Testable Architecture for Implementing Self-Organization Mechanisms **43**
- 4.1 Related Work 45
- 4.2 The Corridor Enforcing Infrastructure: An Architectural Pattern for Self-Organizing, Adaptive Systems 46

4.3 Failure Definition and Categorization of Self-Organization Mechanisms 48
4.3.1 Weaker Notion of Correctness for Self-Organization Mechanisms: Definition of Failure . . . 48
4.3.2 Boundaries of Self-Organization: Tolerable and Intolerable Environmental Faults . . . 50
4.4 Prerequisites and Benefits for Testing Based on the Corridor Enforcing Infrastructure . . . 51
4.4.1 Gain for Testing Based on the Corridor Enforcing Infrastructure 52
4.4.2 Realizations of the Concepts of the Corridor Enforcing Infrastructure: Application Cases . . . 52

5 Isolating and Integrating Self-Organization Mechanisms for Testing 55
5.1 Related Work . . . 57
5.2 Disassemble and Isolate Self-Organization Mechanisms . . . 59
5.3 Reassemble Self-Organization Mechanisms . . . 61
5.4 Test Architecture for Isolated Testing of Self-Organization Mechanisms 62

6 Closed-Loop Model-Based Testing for Continuous and Discrete Self-Organization Mechanisms 67
6.1 Related Work . . . 69
6.1.1 Run Time and Design Time Approaches for Testing Adaptive Systems . . . 70
6.1.2 Model-Based Testing . . . 71
6.1.3 Back-to-Back Testing . . . 73
6.2 Closing the Loop of Model-Based Testing . . . 73
6.2.1 Feedback in Model-Based Testing . . . 75
6.2.2 Concept of Run Time Models . . . 77
6.2.3 Model Reflection for Reflecting Changes in the System under Test 78
6.3 Probabilistic Models for a Continuous Self-Organization Mechanism . . 79
6.3.1 System Model . . . 79
6.3.2 Environment and Test Model . . . 81
6.4 Fault-based Testing Models for Discrete Self-Organization Mechanisms 85
6.4.1 The System Model for Discrete Self-Organization Mechanism . 86
6.4.2 The Environment and Test Model for Discrete Self-Organization Mechanisms . . . 86
6.4.3 Designing Test Models with Environment Faults . . . 88
6.5 Back-to-Back Testing of Test Model and Implementation . . . 90
6.5.1 Using Executable Run Time Models for Back-to-Back Testing . 91
6.5.2 The Special Case of Back-to-Back Testing Self-Organization Mechanisms . . . 91
6.6 Evaluation . . . 92
6.6.1 Production Cell—Testing an Integrated, Discrete Self-Organization Mechanisms in a Back-to-Back Test Setting . . . 94
6.6.2 Energy Grid—Testing a Disassembled, Continuous Self-Organization Mechanism . . . 103

6.6.3 Load-Balancing Web-Service—Evaluating the Test Approach in a Controlled Experiment . . . 117
6.6.4 Pill Production—Investigating Reusability and Generalizability of the Test Model in Resource-Flow Systems . . . 121
6.6.5 Apache Hadoop—Testing an Industrial Case Study in Full Integration . . . 123

7 Test Case Generation for Flat-Branching Test Problems 133
7.1 Related Work . . . 135
7.1.1 Search-Based Test Case Generation . . . 135
7.1.2 Adaptive Test Automation . . . 136
7.2 Boundaries of Self-Organization Mechanisms: A Boundary-Interior Test Case Generation Approach . . . 136
7.2.1 Boundary Interior Test Case Generation for SO Mechanisms via Search-Based Testing . . . 138
7.2.2 Heuristic-Based Selection Strategy for Automated Online Test Case Selection and Reduction . . . 139
7.3 Adaptive Test Cases to Enable Reasoning During Test Execution . . . 140
7.3.1 Annotating the Purpose a Test Case for Enabling Self-Reflection 141
7.3.2 Outlook: Planning Optimal Rule Instantiations by Optimizing Diversity of the Test Cases . . . 143
7.4 Evaluation . . . 144
7.4.1 Production Cell—Boundary-Interior Test Case Generation . . . 144
7.4.2 Load-Balancing Web-Service—Adaptive Test Case Execution . . 148

8 Performance Testing for Self-Organization Mechanisms 151
8.1 Related Work . . . 152
8.1.1 Metrics for Adaptation Mechanisms . . . 153
8.1.2 Metrics for SO Mechanisms . . . 159
8.2 Requirements for Performance Metrics for SO Mechanisms . . . 159
8.3 A Distributed Performance Metric for SO Systems . . . 160
8.3.1 Time Performance of SO Mechanisms . . . 160
8.3.2 Quality Performance of SO Mechanisms . . . 161
8.4 Performance Evaluation Framework . . . 162
8.4.1 Generating Unbiased Evaluation Runs . . . 163
8.4.2 Modeling the Environment for Evaluating the Performance of SO Mechanisms . . . 164
8.4.3 Integrating the Evaluation Sequence Selection in the Evaluation Framework . . . 165
8.5 Evaluation . . . 166
8.5.1 Production Cell . . . 166
8.5.2 Energy Grid . . . 170

9 Conclusion and Outlook 173
9.1 Summary of Research Contributions and Evaluation Results . . . 173

9.2 Open Research Challenges and Future Directions 176

Bibliography **179**

1

Overview and Motivation

For many years, nature has been a source of inspiration for the development and design of software systems. The examples range from optimization, predictive modeling, multi-agent systems, to self-organizing systems. The aim is to learn from biological systems how to cope with complex situations and problems by imitating their behavior, structure, and concepts of mechanics. For instance, it is possible to solve an optimization problem by the concepts of genetic selection: the feasible solutions are depicted as genomes and by recombining and evaluating generations an optimal individual is selected [82]. Another problem to be solved analogously to nature is addressed by Self-Organizing, Adaptive System (SOAS): the complexity of managing large-scale systems is handled by adapting the system's parametrization or even structure with simple local rules. The local rules are executed autonomously by the system itself to adapt to a newly arising situation. Adaption concerns the change of the system's parameterization whereas Self-Organization (SO) changes the system's structure itself. The imitated SO behavior is observed at the folding of proteins, the flocking behavior in birds and fish [28], or even in society [85]. Ashby [11] first discussed the application of SO concepts in software (or more generally in a machine). An SO mechanism, in a software system, is the inherent part of the system that enables it to induce the change of the organization resp. structure of the entire system or parts of the system during its execution, i.e., at run time. Serugendo et al. [146] are differentiating between strong and weak SO, where strong means the SO mechanism is not centralized and weak SO implies a central SO mechanism. Self-Organization is designed to cope with an ever-changing environment. The dynamic arises from the lack of knowledge or just from the hard to predict operational situations of the system at run time. This uncertainty is often a consequence of complex systems, i.e., a system composed of many different (often heterogeneous) components that interact with each other. These complex systems are becoming ubiquitous, e.g., in cloud computing, the internet of things, industry 4.0, or big data. Self-Organization strives to offer new flexibility, that can handle the complexity, inspired by nature.

However, this new flexibility comes at a price: the design and implementation of an SO mechanism that is reliable[1] is a demanding engineering task. The primary objective of engineering software, according to Mills [103], is *"the production of programs that meet specifications"*. The task can be split into two major task: (1) the systematic development and implementation of the SO mechanisms and (2) the systematic assurance that the

[1] We follow the reliability definition by Naresky [112], that is specified as follows: *"The ability of an item to perform a required function under stated conditions for a stated period of time.'*

developed program meets its specification. Engineering SO mechanisms is challenging by nature: A SOAS uses SO in order to handle dynamic conditions by adapting the system's organization at run time, allowing for handling an uncertain and ever-changing environment. The dynamic, however, implies underspecification at design time, i.e., the system's behavior is not fully specified as not all possible situations are foreseeable at design time. This challenges software engineering in its fundamentals. These challenges have been identified in the research community [40, 146]: different communities[2] have been established that focus on investigating the development and implementation of SO mechanisms. However, less effort has been put into the verification task: the assurance that the developed program meets its specification. Indeed, verification is needed to engineer reliable SO mechanisms thoroughly. Thus, there is a gap between the advancements made in the development and implementation of SO mechanisms and the advancements made so far for the verification of SO mechanisms. This thesis is narrowing the gap by contributing to the state of the art in software engineering for SO mechanisms with an Model-Based Testing (MBT) approach.

1.1 Key Challenges in Testing Self-Organization Mechanisms

Testing is the effort of executing programs with the intention of revealing failures[3] [108]. For this purpose, the program is challenged by test cases, containing specific inputs, execution instructions, as well as expected outputs, amongst others. Thus, it is possible to compare the actual output with an expected output of every test case after it is executed. Revealing a failure is achieved if both outputs differ. In an idealistic case, all failures are revealed—and their causing fault is removed—before releasing the software. For this purpose, every possible test case for the SuT has to be generated (either manually or automated) as well as executed and evaluated; this is called exhaustive testing. However, this effort is infeasible for large-scale programs [121]. In order to mitigate that problem, only a subset of all possible test cases is executed on the SuT. The subset selection is problematic since is not always easy to determine which test cases to select. The test cases that should be picked are the ones that are able to reveal the failures of the SuT. However, this selection criterion is only possible to be evaluated, if the test case has been already executed. Different theories [170, 177] have investigated that problem. Most approaches in practice accept a specific inaccuracy, in an optimistic sense [121]. The inaccuracy is applied by only executing test cases that follow a certain adequacy criterion, that might or might not cover any possible failure. Nevertheless, if the adequacy criterion is achieved, the SuT is optimistically assumed as thoroughly

[2] The most important communities that share the investigations of SOAS are, according to Google Scholar h5-index and h5-median rating, meeting at the following annual conferences with SOAS as a central theme: the *International Symposium on Software Engineering for Adaptive and Self-Managing Systems* and the *IEEE International Conference on Self-Adaptive and Self-Organizing Systems* together with numerous workshops and journals, e.g., the *ACM Transactions on Autonomous and Adaptive Systems*.

[3] This thesis is following the common definition of error, fault, and failure, as proposed by Naik and Tripathy [111] and the IEEE standard 610.12-1990 [77]: The situation where the specified behavior is not corresponding to the actual behavior is called a failure. This failure is caused by an error, that is a state in the System under Test (SuT) which might lead to failure. An error is caused by a fault, e.g., a human programming fault.

tested. The adequacy criterion defines the thoroughness of the test suite, the collection of all test cases, according to different test requirements, e.g., code coverage. Thus, one key challenge in software testing is the **generation of adequate test cases for the test suite**.

Indeed, the compiled test suite needs to be executed and evaluated in order to reveal failures. For this purpose, a test scaffold is needed. A test scaffold is the execution environment that is composed of test driver and test stubs. A test driver enables to provide the input, as specified in the test cases, to the SuT and the test stub is substituting the demanded components of the SuT that are not present. **Designing and implementing the test scaffold** is a further key challenge in software testing.

Once a test suite has been executed, all test cases need to be evaluated, i.e., a judgement is needed, whether a failure has occurred during or after the execution of any test case. A test case can be evaluated right after its execution using the current state of the SuT. Further, all test cases can be evaluated at once after the execution of the complete test suite, based on the logs. There are different ways for carrying out the evaluation, but in all cases, a ground truth needs to be given. The ground truth provides the information of the expected outcome of a test case. The provision of this ground truth is referred to as a test oracle. The oracle might be provided for every test case separately, i.e., along with the generation of the test case, an expected outcome is generated. Alternatively, an automated oracle is provided, i.e., a program that returns an expected value for given input, that is often referred to as a "gold standard". The automated test oracle eases the task of automated test case generation and increases the overall quality of the test process, but is demanding to be supplied. An automated test oracle allows for only test input generation, i.e., no expected output is given in a test case, since the evaluation is carried out by the oracle. Further, an automated oracle is less prone to errors as a manual oracle [13], thus, increases the test process quality. The **provision of the test oracle** a key challenge in testing.

These three presented key challenges need to be addressed in every testing endeavor.

The properties of the SO mechanisms like *inherent non-deterministic behavior, an ever-changing environment, a high number of interacting components, and interleaving operations* make it hard to achieve a systematic testing approach for SO mechanisms that is coping with these challenges. The key to success is automation since it is rarely possible to cope with the high number of demanded test cases (that are necessary as a result of the vast state space) manually. However, the automation of testing SO mechanisms is faced by the complexity of the system class requiring techniques to cope with the following additional key challenges, specific for SO mechanisms:

Error Masking SO mechanisms—and in general SOAS—are designed to be robust and flexible under ever-changing environmental conditions. As a side-effect of these properties, an SO mechanism itself, other SO mechanisms, or adaptation mechanisms might cover the tracks of possible failures that should be revealed while testing the SO mechanisms. Thus, the incorrect behavior of one SO mechanism could be compensated by another mechanism masking the failure.

For instance, assume an erroneous SO mechanism that returns wrong or inappropriate system structures as a result to the controlled components. This fault then could be masked by an adaptation mechanism of the components that compensates the wrong system structure by a high and costly amount of adaptation. The SO mechanism would encompass a fault, but no failure is visible at first glance since the adaptation mechanism masks it by keeping the system alive.

Isolate SO mechanisms are based on interaction with the system's components. In the majority of cases, several different SO mechanisms are incorporated; their overlap of interaction with the components leads to so-called interleaved feedback loops [159]. These are challenging in testing because it is hard to get dedicated results for a single SO mechanism. In order to address this challenge, there is a need for isolated testing of single SO mechanisms.

As an example of this challenge, assume an SO mechanism that forms organizational structures, e.g., by partitioning the system's components. A further mechanism, however, performs its calculations for parametrization of the components based on this structure, e.g., for forming an evenly distributed share of workload for each partition. These two mechanisms are interleaved, and a dedicated test result for the mechanism performing its calculation on the structure is only possible if they are isolated, since the results depend on the results of the first and vice versa. However, this isolation is hard to achieve due to the high dependencies between the two mechanisms.

Test Oracle The oracle problem is a well-known challenge for all testing endeavors [17, 121]. However, the properties of SO mechanisms increase this problem: For classical testing, the conditions of execution for the SuT, as well as the particular requirements, are known. Let us call these facts the "known-knowns".[4] For Self-Organization Mechanism under Test (SOuT) we know that there are unknown conditions of execution where we can hardly decide a priori, i.e., at design time, whether which transition from one state to another is correct or not; we call these conditions the "known-unknowns". Moreover, for the SOuT there might even be situations we are not aware of at all, which we call the "unknown-unknowns". An oracle that is capable of evaluating the test results of an SO mechanism at least has to be able to handle the "known-unknowns".

For an instance of "known-unknowns" consider a smart energy grid setting where different power plants are self-organized in different so-called autonomous virtual power plants. If weather-dependent power plants are included, the SO will depend on the weather conditions. We know that there are different conditions like sunny, rainy, windy, etc., but we also know that we do not know all different possible combinations with the correct organizational structures at design time of the test (or at least we cannot compute all). However, an oracle has to cope with that and has to decide whether a result is accepted as correct or rejected as incorrect.

Branching State Space A vast state space is a common challenge for software testing [17], but, as for the oracle problem, SO mechanisms add a further dimension. Most

[4] The classification of known-knowns, known-unknowns, and unknown-unknowns is borrowed from United States Secretary of Defense Donald Rumsfeld's response given to a question at a U.S. Department of Defense news briefing on February 12, 2002.

of the approaches in software testing that are coping with a huge state space make use of the structure of the state space in order to reduce the number of test cases needed to be executed. For instance, an infinite loop in a code fragment means an infinite state space, but its ramification degree is rather small. A mechanism applied here is boundary-interior-testing [121] that cuts deep branches at specific lengths. SO mechanisms, however, are mostly based on heuristics for coping with the ever-changing environment, making the result non-deterministic; this leads to a wide and a rather flat-branching structure of the state space, making the most of the classical techniques hardly applicable.

1.2 Model-Based Testing for Self-Organization Mechanisms

Addressing the challenges for testing SO mechanisms starts with a new notion of a failure. This new notion is needed due to the underspecification of SO mechanisms. Self-Organization mechanisms' behavior is not entirely specified in order to allow for adaptation and SO at run time. Nevertheless, for testing we at least need to distinguish a situation as correct and not correct; this is possible due to the Corridor of Correct Behavior (CCB) approach [70]. The CCB is constraining the state space of the system that is under the control of the SO mechanism; it defines correct states and incorrect states. A SOAS is allowed to be in an incorrect state, but the SO mechanism has to identify this situation and reconfigure the system (if possible) so that a correct state is reached. For instance, a task allocation to robots is correct if all robots are capable of executing their assigned task and it is incorrect if a necessary tool is broken. The CCB does not specify the correct transitions between the states, but only describes the acceptable states in the form of a corridor. In this thesis, the CCB is used for supplying a notion of failure for SO mechanisms as well as making SO mechanisms testable.

Testability is a property that is due to an architecture of a system. Most of the engineered SO mechanisms are equipped with an architecture that is built upon feedback loops [22]. The Corridor Enforcing Infrastructure (CEI) is a pattern for SO mechanisms that is explicitly describing this architecture and provides testability. We will further show that an implicit CEI, i.e., a feedback-loop-oriented SO mechanism, is also testable on that basis if the test engineer can identify the detection, computation, and distribution component of an SO mechanism. As discussed by Brun et al. [22], this feedback-loop-oriented design of SO mechanisms is a wide-spread realization of SO.

The phases of SO in the CEI are detection, computation, and distribution. Based on these phases, it is possible to disassemble and isolate the SO mechanism. Isolation is not only a matter of handling the test problem by the divide-and-conquer principle, it also addresses the fact, which the full integration masks errors in the integrated SO mechanism. The CEI pattern is here again the foundation for isolation and also for integration of the SO mechanism. However, the fact that the CEI does not need to be explicitly implemented by the SO mechanism, as long as the SO mechanism is based on a feedback-loop-like design.

The MBT concept–which is a well-established approach for testing complex system [17, 21, 68, 126, 162]—is central for the presented approach for testing SO mechanisms.

Model-Based Testing is based on the idea of making the implicit knowledge of the test engineer(s) (concerning the SuT and its intended behavior) explicit [17, 108]. Thus, MBT is structuring a test approach and its activities, which is allowing for automation of testing activities. These activities encompass, in this thesis, test case generation, test cases execution, and test case evaluation. Indeed, this is a complex task resulting from the complex SuT, as described in the challenges above. The MBT approach allows for abstraction, enabling to handle this complexity. We will demonstrate how abstraction allows for describing the SOAS, its environment, the SO mechanism as well as all interdependencies between them. The model is used as a source of information provided by the test engineer that is used, amongst other things, for test automation. However, for SO mechanisms, not all information can be provided at design time. This lack of information is mainly due to the complexity of the system and its environment, that is handled by abstraction. However, also due to the inherent non-determinism of the SO mechanism itself. The information by the test engineer at design time needs consequently be completed by information from the run time of the test to allow for testing SO mechanisms. However, nowadays MBT are directed one-way: from the requirements to the completed test project. This approach is insufficient for testing SO mechanisms. In this thesis, MBT is extended by introducing feedback for establishing closed-loop MBT. This is enabled by the concept of executable run time models. These models are capable of changing their information content, i.e., their structure and their instances, by reflecting changes of the subject that is described in the model. By bringing this capability into MBT, it is possible to handle the run time behavior of SO mechanisms. A further aspect that is exploited is the ability to execute the models, that are used for generating test inputs by simply executing every part of the model that is used as a stub and bringing the data to the driver. Thus, it is possible to use all information of the model and generate test cases. Decisive for the quality of the testing process is the quality of the test model. This difficult engineering task is supported by the Back-to-Back (BtB) testing approach that is developed in this thesis. Here, a test and a development engineer are developing the test model as well as the implementation of the SO mechanism back-to-back and use both for testing. This process is carried out in an early development stage to assure a correct interpretation of the requirements.

We will use five different case studies to evaluate the concepts and methods. The evaluation contains systems with an implicit as well as an explicit CEI. We will show that in both cases, testing is enabled by the CEI's concepts. The abilities of the MBT approach of revealing failures, real ones as well as injected ones, are demonstrated in these different case studies. Further, the test case generation concepts are extended by a search-bases approach. The approach can handle the flat-branching test problem of SO mechanism. Flat-branching is a result of the multiple different possible states that are available for a SOAS as well as the multiple decisions options of the SO, making the state space flat-branching instead of deep-branching. This different state space made the new test case generation concepts in this thesis necessary. The evaluation will show that a speed-up of revealing failures compared to random testing up to facto 500 is possible to achieve.

Besides testing functional requirements, the MBT approach presented in this thesis allows further for testing non-functional aspects as well. We will discuss the performance of the SO mechanism and will show how it is possible to do this in an MBT fashion.

1.3 Main Contributions and Thesis Outline

Testing SO mechanisms is challenging due to error masking, isolation, and integration of highly interwoven mechanics, the undefined notion of a failure, and the branching state space. Indeed, testing also demands for adequate test suite generation methods, and suitable test scaffold, and the provision of a test oracle for the specific test problem. The contribution of this thesis is addressing all these challenges for SO mechanisms within nine chapters. Figure 1.1 shows the structure of the presentation of the thesis. The contributions, as well as the structure of this thesis, are outlined in order to guide through this thesis. The thesis covers five different case studies, that are introduced in Chapter 2. The case studies are used throughout the thesis for exemplifying the presented approaches. Further, each case study is implemented and available for testing. In the evaluation, these implementations are used to demonstrate the capabilities of the made approaches. Each case study uses different SO mechanisms that have been proposed by different authors and institutions resp. industries. The contributions made throughout this thesis are clustered into following six parts, which show the key findings made, which have already been published in 28 peer-reviewed scientific publications that have been co-authored by the author of this thesis.

Establishing a Notion of Failure for Self-Organization Mechanisms

Self-Organization mechanisms can restore the SOAS from a situation where it is no longer able to fulfill its duties by reconfiguring and adapting the structure of the system. Judging a failure is difficult if the system might be able to recover from a situation where temporarily the specification is not fulfilled (what qualifies as a failure from the known notion [108]). This thesis offers a new notion of failure for SO mechanisms, that is coping with the specific behavior of SO mechanisms. This contribution is described in Chapter 3 and is used for the next chapters as an input.

Establishing Testability for Self-Organization Mechanisms

Self-Organization mechanisms are highly interwoven with the controlled system and its environment. This characteristic makes it difficult to control and observe the behavior of SO mechanisms, what is known as testability. With the architectural pattern of the CEI, this thesis offers a generic approach for making SO mechanisms testable. The SO mechanisms are controllable by having defined phases of SO (in a feedback-loop-oriented process) and defined interfaces. These are also allowing for observability. Chapter 5 is presenting a testable architecture for SO mechanisms, the CEI. Further, it is shown which system class of SO mechanisms can be tested on that basis. We, therefore, distinguish whether the CEI is implemented explicitly or implicitly. Both can benefit from the approach. The implicit implementation is wide-spread in the design of SO mechanisms.

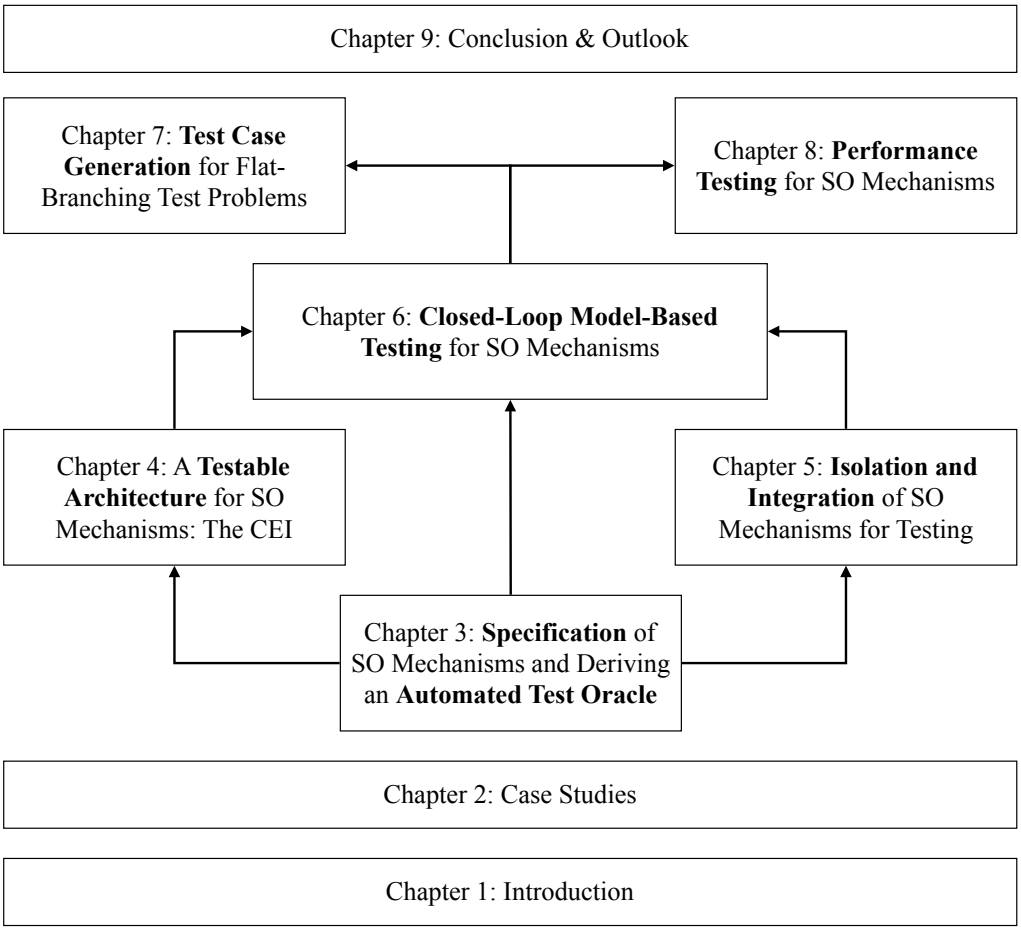


Figure 1.1. This thesis is structured in the shown nine chapters. At the beginning the topic is motivated and introduced. Afterwards, five different case studies are explained that are used throughout the thesis for exemplifying the presented approaches and their evaluation. The foundation for testing is the specification, presented in Chapter 3. The content is delivering input for the following chapters (Chapters 4 to 6). The main contribution is enabling MBT, described in Chapter 6. As conceptual input, Chapters 4 and 5 are delivering the foundation for testability, isolation, and integration of SO mechanisms. In Chapters 7 and 8, the MBT approach is completed by search-based test case generation concepts and a performance testing approach.

Enabling to Isolate and Integrate Self-Organization Mechanisms for Systematic Testing

The CEI also enables to isolate and integrate SO mechanisms. This thesis shows how SO mechanisms are systematically decomposed and assembled for isolated and integrated testing. The contribution is set into a test architecture in Chapter 5, offering a complete test scaffold for SO mechanisms.

Establishing a Closed-Loop Model-Based Testing Approach Suited for Testing Self-Organization Mechanisms

The realization of the test architecture is done within Chapter 6 in an MBT approach, that is designed and suited for SO mechanisms. The characteristics of SO mechanisms make it necessary to cope with the adaptive behavior at run time. For this purpose, this thesis offers an extension of the known MBT approach by implementing feedback loop and using so-called run time models to reflect the SuT and its environment and enabling to incorporate the current state of the SuT into the test model. Within the MBT approach, two different kinds of SO mechanisms, namely, continuous and discrete ones, are addressed with a suitable modeling approach. The contribution includes a probabilistic and a fault-based test model. The evolution showed the capabilities of the modeling approach for revealing failures.

Establishing a Search-Based Test Case Generation Approach for Efficient Testing of Self-Organization Mechanisms

A search-based test case generation approach is presented in Chapter 7 to complement the MBT approach by an efficient way of generating test cases. The contribution enabled to reveal the same kind of failures up to 500 times faster than in a random testing approach.

Extending the Model-Based Testing approach for testing the performance of Self-Organization mechanisms

The MBT approach for testing is able to cope with functional tests, but also offers the ability to test non-functional aspects. In Chapter 8 an approach for measuring and testing the performance of SO mechanisms is presented. This contribution includes a definition of performance for SO mechanisms as well as a complete approach for automated testing of this performance indicators.

Summary. Five SOASs from a range of application domains are introduced in this chapter. They serve as case studies for the remainder of this thesis and are used for the evaluation of the approaches proposed in this thesis. The presented SOASs are categorized by the form of SO that is performed in the system. We use the environment of the SO mechanism for that categorization. Either continuous environmental properties or discrete ones are determining the environment of an SO mechanism; that leads to different requirements for SO mechanisms. Thus, we differentiate between continuous and discrete SO mechanisms. Each case study is an implemented system ranging from research prototypes to a large open source system widely used in industry. We discuss for every case study the application case of SO, the challenges for testing, and the implementation details. The case studies have been described, modeled and tested in [51–54, 57], amongst others.

2 Case Studies

2.1 Case Studies with Continuous Self-Organization Mechanisms 12
2.1.1 Decentralized Power Management 12
2.1.2 Self-Adaptive Apache Hadoop Manager 15
2.2 Case Studies with Discrete Self-Organization Mechanisms 17
2.2.1 Self-Organization Production Cell 17
2.2.2 Self-Organized Personalized Medicine Pill Production System 20
2.2.3 Self-Adaptive Webservice System: ZNN.com 22

We introduce and discuss a set of different case studies throughout this chapter. The case studies are used as running examples as well as for a thorough evaluation of the presented concepts. The presented case studies that we use for testing are systems that are already implemented. The systems range from multi-agent-based implementations to service-oriented ones and the Self-Organization (SO) paradigms implemented are centralized (weak SO, according to Serugendo et al. [146]), decentralized (strong SO, according to Serugendo et al. [146]), or regio-centralized SO [6] . Thus, the generality of the approach for testing SO mechanisms can be demonstrated.

Self-Organization—A Definition The term SO is not always defined in a unified way. Self-Organization, as we consider it, follows the definition by Serugendo et al. [146]: The reconfiguration of a system, or a subsystem, with the intention to restore, or even increase, the productivity, in sense of fulfilling the functional obligations, at run time under ever-changing environmental conditions without explicit external control. All case studies introduced in this chapter are using this form of SO, but the mechanisms are named by the authors and developers as self-adaptive or adaptive systems. We will stick to the original labeling by the authors and developers.

Classification of Self-Organization In general, all investigated SO mechanisms are highly dependent on the environment of their controlled system. The environment is everything, which is either directly or indirectly influenced by or influencing the Self-Organizing, Adaptive System (SOAS). It determines the need for SO, the way of SO, as

well as the capabilities of the SO mechanism. In general, the domain, as well as the solution space of the SO mechanism, is formed by the properties of the environment. Continuous or discrete properties constitute this space. We use this as a characteristic of the SO mechanism to be investigated. The case studies presented in this chapter are divided into these two categories.

We introduce a decentralized power management system, that is forming a continuous input and output space. This system incorporates the SO mechanism to organize and reconfigure the structure of the system, that is organized into small units working together. The second system of the class of continuous SO mechanisms is embedded into Apache Hadoop, a system implementing the MapReduce paradigm. Within the system, the SO handles the job allocation by adapting the system to volatile tasks. For the class of discrete SO mechanisms, we introduce two case studies that are resource flow systems. The resource flow is first established in a production environment, producing small and fast-changing batch sizes. Second, the resource flow is formed by a pill and medicine arrangement system, producing individualized medicine. Both share an underlying meta-model for resource-flow oriented SOAS. Similar SO mechanisms can be used in both applications. Finally, the third system with discrete SO mechanisms is a system for resource balancing in a cloud-like environment. The resource balancing is achieved via SO of the servers in the system.

2.1 Case Studies with Continuous Self-Organization Mechanisms

Continuous SO mechanisms are characterized by their domain and solution space, that is described by properties that are continuously changing their value (this is similar to the mathematical continuous-state definition, cf. Cassandras and Lafortune [29]). The SO is thus triggered by thresholds, not by discrete events. The SOAS is, in contrast to a discrete SO mechanism, never stopped for the reconfiguration by the SO mechanism; the system continues to evolve. Thus, it is still able to fulfill its duties to some extent in the moment of a threshold violation. However, the SOAS is not able to achieve its intended level of quality in this situation. The threshold violation might even lead to a critical situation for the SOAS. During the reconfiguration by the SO mechanism, the system continues to evolve while a new configuration is computed and distributed.

We introduce a case study of that class: a decentralized power management system. The SO in that system controls the structure of the system which is formed by decentralized units. In this section, the system, in general, is introduced as well as the SO that is needed. The concrete implementation of different SO mechanisms is described in the evaluation section where these particular SO mechanisms are evaluated. We will further summarize the multi-agent system based implementation setting of the SOAS here.

2.1.1 Decentralized Power Management

The widespread installation of weather-dependent power plants as well as the advent of new consumer types like electric vehicles put much strain on power grids. Additionally, small dispatchable power plants, e.g., biogas plants, owned by individuals or

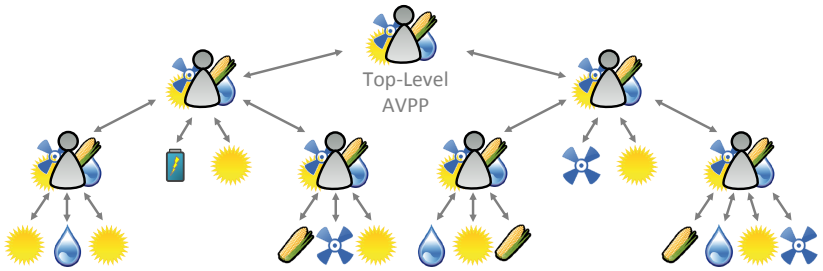


Figure 2.1. Hierarchical system structure of a future autonomous and decentralized power management system, taken from [52]: Power plants are structured into systems of systems represented by Autonomous Virtual Power Plants (AVPPs) that act as intermediaries to decrease the complexity of control and scheduling. AVPPs can be part of other AVPPs. The left child of the top-level AVPP, for instance, controls a solar power plant, a storage battery, and two subordinate AVPPs.

cooperatives[1] feed in power without external control. Current plans are therefore to scale the controllable output further by installing additional flexible dispatchable power plants and to drive the expansion of the power grid forward. To save expenses, gain more flexibility, and deal with uncertainties, future *autonomous* power management systems have to take advantage of the full potential of dispatchable prosumers[2] by incorporating them into the scheduling scheme. Further, aleatoric uncertainties have to be anticipated when creating schedules and compensated for locally to prevent their propagation through the system.

To meet the challenges of future power management systems, Steghöfer et al. presented the concept of AVPPs in [152] (similar visions of virtual power plants are discussed in [129]). AVPPs represent SO groups of two or more power plants of various types (cf. Figure 2.1). The organizational structure represents a *partitioning*, i.e., every power plant is a member of exactly one AVPP, which is established and maintained by a (partitioning-based) SO algorithm, the part of the SO mechanism responsible for computing new configurations for the system under control. Constraints that specify valid partitionings, e.g., a maximum number of power plants that may belong to one AVPP or that every power plant has to belong to exactly one AVPP, among others, induce the requirements for the SO mechanisms over the space of all partitionings. In this setting, each AVPP has to satisfy a fraction of the overall demand. To accomplish this task, each AVPP autonomously and periodically calculates schedules for directly subordinate dispatchable power plants. Further, each AVPP's dispatchable power plants have to reactively compensate for deviations resulting from the local output or load fluctuations, i.e., uncertainties, to avoid affecting other parts of the system.

AVPPs autonomously adapt their structure, i.e., are self-organized, to changing internal or environmental conditions. They can live up to the responsibility of maintaining an organizational structure enabling the system to hold the balance between energy supply

[1] A cooperative is a jointly owned power plant engaging in the production of energy, operated by its members for their mutual benefit, typically organized by farmers.
[2] We use the term "prosumer" to refer to producers as well as consumers.

and demand. In particular, if an AVPP repeatedly cannot satisfy its assigned fraction of the overall demand or compensate for its local uncertainties, i.e., output or load fluctuations locally, it triggers a reorganization of the partitioning. The goal is to form homogeneous partitionings in the sense of a structure of similar AVPPs that are likely to feature a heterogeneous composition: On the one hand, by distributing unreliable power plants among AVPPs, the chance of fluctuations is reduced, and the system's robustness increases. On the other hand, by balancing the AVPPs' degrees of freedom, e.g., by mixing different generator types, their ability to deal with uncertainties, i.e., fluctuations, locally is promoted. To cope with the vast number of dispatchable power plants, the concept of AVPPs proposes a scalable, hierarchical structure in which AVPPs act as intermediaries. This system decomposition reduces the number of dispatchable power plants (including directly subordinate AVPPs) each AVPP controls resulting in shorter scheduling times for each AVPP and the overall system. Thus, the implemented SO can be classified as strong SO. The environmental aspects described are forming the typical environment of continuous SO mechanisms.

Challenges for Testing

In the smart-grid application, we have to cope with successfully isolating the SO mechanisms for testing them in the loop. Further, we need to reduce error-masking, i.e., the self-healing aspects of the AVPP structure are likely to mask an error in some components by merely compensating it, since the partitioning algorithm and the mechanism that balances supply and demand in each AVPP influence each other significantly. Error masking can also occur in the partitioning algorithms if, e.g., the result of the SO algorithm is faulty, but the system state after the reorganization process is valid. The problem of state space explosion has to be tackled in this case study due to the stochastic nature of the partitioning algorithms to deal with the large search spaces and the stochastic environment, e.g., weather conditions, market prices. Finally, the testing oracle problem occurs in this context since it is hardly decidable at design-time which partitioning for which power plants is correct as this depends on the unknown environmental setting of the controlled power plants as well as on the unknown states of the autonomous power plants themselves.

Implementation Details

As we focus on the self-organized creation of partitionings for the evaluation of the approaches presented throughout this thesis, without loss of generality, we only regard a "flat" structure of AVPPs in the following. In this structure, power plants self-organize into a single layer of AVPPs that resides directly below the root, i.e., the top-level AVPP. In Chapter 6, we present the algorithms called SPADA and PSOPP, for the self-organized creation of partitionings. We use a Java implementation of the energy grid management system that is based on a multi-agent system called TEMAS [9]. The TEMAS is based on a stepwise execution model out of the box, which allows monitoring consistent states of the system at specific points in time.

The implementation has been carried out in the DFG Research Group OC-Trust and published in [9, 152] and supplied by the authors for testing.

2.1.2 Self-Adaptive Apache Hadoop Manager

Hadoop is one of the most popular and wide-used software platforms for big data processing applying the MapReduce paradigm to a large number of different applications and workloads. MapReduce is a programming paradigm for processing and generating huge data sets [41]. Dean and Ghemawat [41] developed it as a unified way to process large amounts of raw data, e.g., crawled documents or logs, to compute different kinds of derived information from the raw data. Dean and Ghemawat [41] designed MapReduce as a new abstraction allowing to express rather simple computations that are performed on the raw data, while the complexity and details of parallelization, fault-tolerance, distribution, and load-balancing are hidden. The abstractions used by MapReduce have been inspired by the map and reduce primitives in functional programming languages like Lisp. Using MapReduce requires to define a map operation for each logical record in the input data set to compute a set of intermediate key/value pairs. Afterward, a reduce operation for processing the raw data is applied to all values with the same key, which is resulting in the derived data. Using the MapReduce approach, it is possible to utilize the resources of large distributed systems and consequently to power up data processing, even if the programmer is not experienced with parallel and distributed systems [41].

However, the actual gain of the application for effectively distributing MapReduce jobs within a cluster depends on the configuration of a bunch of complex parameters which need to be tuned for a specific task or workload. The *YARN* (*Yet Another Resource Negotiator*) resource manager is the component of Hadoop which is responsible for scheduling and controlling the workload within the cluster. The parameterization of the *YARN* controller is decisive for the job's performance. The best practice for setting the parameter is a best-effort configuration that is based on experience or static profiling, relying on apriori knowledge about the job (cf. [31, 75, 80]). Zhang et al. [178] developed a self-adaptive component on top of the *YARN* resource manager. It is a centralized implementation of the *MAPE* architecture (cf. [87]), i.e., a control loop that *m*easures, *a*nalyzes, *p*lans, and *e*xecutes the adaptation of the parameter setting of *YARN*. Zhang et al. [178] showed that they can speed up the Hadoop instance up to 40% in a volatile environment compared to the best effort solution.

Figure 2.2 shows the high-level architecture of Hadoop used for the case study. The whole system is deployed in docker-swarm[3], used for installing the Hadoop cluster in a virtual environment, where different hadoop-compute nodes and a hadoop-controller are installed. The hadoop-compute nodes and the hadoop-controller are connected within the docker-swarm to form a cluster. The hadoop-compute nodes are representing the working nodes of Hadoop, running in a single container with an own NodeManager and DataNode. The NodeManager is responsible for managing the local resources and gathering information by monitoring the memory and CPU usage of the node and logging this information. This information is shared with the ResourceManager. The information of

[3] https://docs.docker.com/engine/swarm/

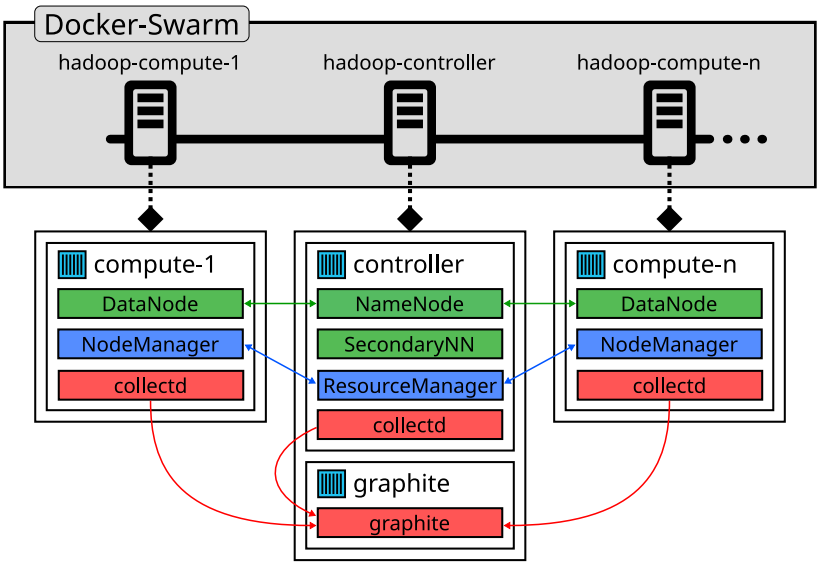


Figure 2.2. Docker-swarm based deployment architecture of a Hadoop instance with the adaptive extension by Zhang et al. [178], supplied by [179].

the node is stored in the DataNode which is part of the functional filesystem. For this purpose, the local DataNode is connected with the NameNode of the hadoop-controller. The SO mechanism is responsible for adapting the configuration of the ResourceManager within the hadoop-controller. The hadoop-controller is also equipped with the NameNode that host the keys with the MapReduce approach, i.e., the file-system index and the number of DataNodes. Besides the controller the hadoop-controller also hosts a graphite container. The graphite node is providing a service for collecting data and information from the Hadoop system, that is used by the SO mechanism. The information is used by the SO mechanism to provide the following three properties at run time [178]:

1. Enhancing the jobs parallelism by adapting the maximum-am-resource-percentage (MARP)[4] value according to the current jobs.
2. Increasing the job throughput by continuously adapting the limiting parameter of the ResourceManager.
3. Preventing large drops of memory utilization caused by the completion of MapReduce tasks by smoothing the input continuously to stabilize the value.

By these responsibilities, that SO mechanism is classified as a continuous SO mechanism.

Challenges for Testing

For testing the self-organizing, resp. self-adaptive, Resource Manager of Hadoop we have to cope with the complexity of the distributed system environment which is the foundation of Hadoop. The isolation of this mechanism is one challenge to overcome to reduce error-masking. Next, test scaffolding and test inputs need to provide complex

[4]The maximum percent of resources in the cluster which is used to control the number of concurrent active applications [61].

data, that is a result of the MapReduce processing paradigm, thus isolating or testing means also to provide complex input values for valuable testing. This also leads to a vast state space since the different possible data inputs to Hadoop and configurations that are the test inputs for the SO mechanism are numerous. The different interdependencies between inputs and configurations are hard to predict for the setting of a volatile environment, making systematic testing challenging. Further, as it is not possible to select the correct configuration for the Resource Manager, in the sense of the functionality of the SO mechanism, we are faced with the oracle problem here.

Implementation Details

Zhang et al. [179] provide their implementation of the SO mechanism integrated in the Hadoop environment via docker[5] images at GitHub[6]. The SO mechanism within the docker image is, as Hadoop, written in Java. The authors permitted us access to the source code for our testing purpose. The docker system is an open source project by the Apache Foundation[7], thus, the source code is also available. The virtual environment of docker enables to set up the Hadoop cluster virtually distributed as well as actually distributed.

2.2 Case Studies with Discrete Self-Organization Mechanisms

Discrete SO mechanisms are controlling a system with a domain and solution space that is formed by discrete properties. Thus, a change of a property might lead to a direct demand of the SO mechanism to reorganize the system. The SOAS might be even halted for that reconfiguration. We investigate two groups of discrete systems with discrete SO mechanisms: self-organizing resource-flow systems and self-organizing load-balancing systems. The resource-flow systems are based on a common meta-model and theory for implementing SO in this system class. The load-balancing system for a cloud application is the third system of the discrete characteristics. Self-Organization is here responsible for organizing the resources and their task allocation at run time.

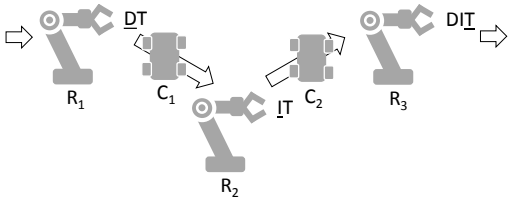
2.2.1 Self-Organization Production Cell

Future production scenarios demand much more flexibility than today's shop floor design to cope with the trend towards small series production, individualized products and the reuse of production stations for different tasks [104]. This flexibility becomes possible due to the increased automation and data exchange in manufacturing technologies. These future cyber-physical systems will integrate SO mechanisms to resolve the tasks of decentralized decision making, to optimize the structure of the system, and to autonomously react to component failures at run time increasing the system's robustness. The self-organizing production cell is the implementation of that vision, where the production stations are modern robots equipped with toolboxes and the ability to change their tools whenever necessary. Figure 2.3 shows an illustration of

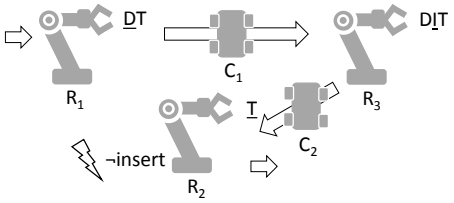
[5] https://docs.docker.com/

[6] https://github.com/Spirals-Team/hadoop-benchmark

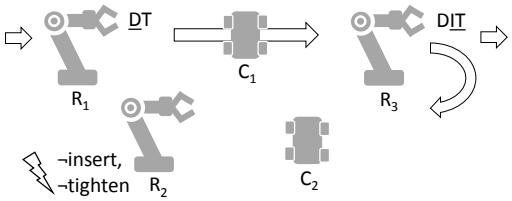
[7] http://www.apache.org



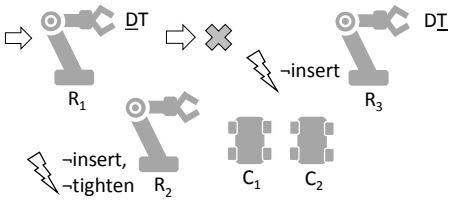
(a) The resource flow starts to the left where R_1 drills a hole into incoming workpieces. Workpieces are successively transported by C_1 and C_2 to R_2 and R_3, applying their insert and tighten tools, respectively. Once R_3 is done, the workpieces leave the system.



(b) After R_2 lose its insert tool, the resource flow is reconfigured: R_2 and R_3 switch roles, each taking over the previous role of the other. The carts transport the workpieces to R_3 before R_2 to guarantee the correct processing order.



(c) After a failure of R_2's tighten tool, R_2 and C_2 are no longer used at all, as R_2 has no more working tools available. Instead, R_3 uses two of its tools on the workpieces successively: Insert, then tighten.



(d) After R_3's insert tool is lost too, the robot cell can no longer be reconfigured such that the task can be completed: There is no robot with an insert tool that can follow R_1's drilling.

Figure 2.3. A schematic overview, taken from [54], of the self-organizing production cell case study with three robots R_1, R_2, and R_3 as well as two carts C_1 and C_2 establishing the resource flow between them. The task is to apply the drill, insert, and tighten capabilities to all incoming workpieces. Each robot's available tools are shown to its right, with D, I, and T denoting the drill, insert, and tighten tools, respectively; the currently allocated ones are underlined. Figure 2.3a shows an exemplary configuration of the robot cell. As depicted in Figures 2.3b to 2.3d, faults result in tool losses that self-organization can cope with by reconfiguring the resource flow; eventually, however, no further reconfiguration is possible as the system runs out of redundancy.

the production cell and its concepts. The robots are connected via mobile platforms, called carts, that can transport workpieces and to reach robots in any order. Thus, the production cell can fulfill any task which corresponds to tools (capabilities) available in the cell. This is possible due to the SO mechanisms that reorganize the carts and robots in a way that the tools are applied to the workpieces in the correct order. Any violation of the calculated configuration at run time triggers the SO algorithm calculating and distributing a new system configuration, as shown in Figures 2.3b to 2.3d .

This characteristic classifies the system as a discrete SOAS. The properties of interest are the availabilities of tools, robots, and carts, their availability is changing discrete from available to unavailable. The implemented SO mechanisms, which are again discussed in greater detail in the evaluation of Chapter 6, are working with these values and even set the system into a hold-mode (called quiescent state) [70, 109, 110].

The self-organizing production cell, as well as the self-organized individualized medicine pill production system, are an instance of the system class of SO resource-flow systems;

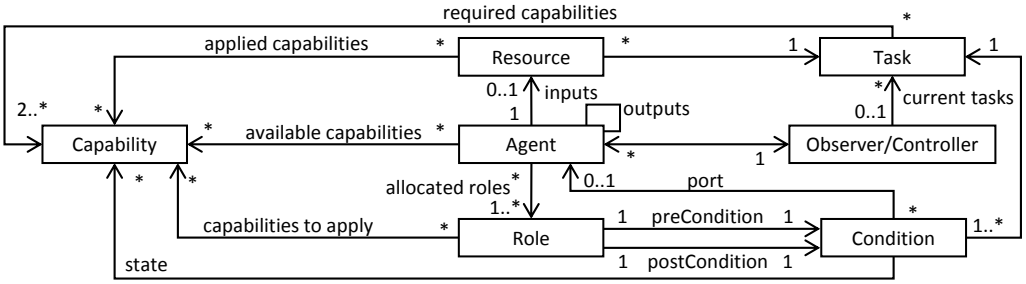


Figure 2.4. A UML class diagram giving a simplified overview of the meta-model for self-organizing resource-flow systems (according to [145]): Resources are passed along a set of Agents, each applying certain Capabilities in order to introduce a Resource into the system (Produce), to remove a Resource from the system (Consume), or to conduct a step towards the completion of the Resource's Task. The Observer/Controller—encompassing the SO mechanism—monitors the Agents and assigns their Roles such that all Resources are eventually fully processed with the correct order of Capability applications. Such a resource flow is specified by the Pre- and PostConditions of all Roles within the system, as well as the inputs and outputs of the Agents that establish their interconnections.

a meta-model [145] for this system class is explained by Figure 2.4. The case study maps to the meta-model as follows: The robots and carts are Agents monitored by the Observer/Controller. The carts transport workpieces, i.e., Resources, between the robots, which have several switchable tools, i.e., Capabilities, such as drills and screwdrivers that they use on the workpieces. A Task requires a workpiece to be processed by a sequence of tool applications, e.g., by applying the drill, insert, and tighten Capabilities. Therefore, the robots and carts are responsible for processing incoming workpieces in a given sequence of tool applications. The Roles assigned to each robot and cart indicate which tools they apply on the workpieces or which robots the resources are transported between, respectively. The Observer/Controller includes the SO mechanism of the system; it is responsible for reconfiguration to compensate for broken tools, blocked routes, or to incorporate new tools, robots, or carts, for instance.

Challenges for Testing

For testing the production cell's SO mechanisms, we have to extract and isolate it from the system. We are going to investigate a centralized as well as a decentralized implementation. Both implementations are tightly interwoven with the system, since the software representations of the robots, carts, and tools, today this is often called digital twin, are all part of the SO mechanism. The Observer/Controller, hosting the SO mechanism in Figure 2.4, is presented in different instantiations an implementations: one Observer/Controller for every software agent in the system. All Observer/Controller paris together are forming the SO mechanism of the system (representing a strong SO mechanims). This functionality must be encapsulated and isolated for testing without changing the behavior of the SO mechanism. That is further necessary to cope with the problem of error masking, as different agents implement the SO mechanism it might be the case that one agent takes over for another and the error will not propagate to a

failure. Indeed, there is also a huge state space to be tested for the SO mechanism since it is formed by the different combinations that constitute the production cell: the concrete number and types (each type with a set of given tools of the available tools) of the robots, carts, and tools as well as different possible production tasks forming the state space of the SO mechanism. Further, the challenge for testing using these discrete concrete properties is also influencing the setting of the test since the changing of an input value corresponds with losing a capability that cannot be restored without resetting the system. Thus, the concrete choose of a test sequence determines the outcome and gives a lot of different chooses. Next, it is challenging to judge over results, since it is hard to decide over every role allocation in every situation at design time.

Implementation Details

Seebach et al. [145] implemented the case study in the so-called Organic Design Pattern Runtime Environment (ORE). The ORE is based on Jadex [123], a Java-based multi-agent system that is implementing the Belief, Desire, Intention (BDI) paradigm for the modeling of agents (cf. Haddadi and Sundermeyer [73]). In general two different SO mechanisms, a central and a decentral one, are available for reconfiguring the system. The SO mechanism has been implemented into a generic Agent, responsible for reconfiguration. This generic agent is coupled to the agents of the system to be self-organized. The implementation has been carried out in the DFG Priority Program Organic Computing in the Project SAVE-ORCA, published in [145], and supplied by the authors for testing.

2.2.2 Self-Organized Personalized Medicine Pill Production System

According to the Personalized Medicine Coalition [120], there is a need for a more efficient and personalized treatment of patients in medicine. To overcome the costly and potentially dangerous trial and error processes to find the right kind of medicine for a patient, the production of drugs must be personalized for each patient. This situation demands individualized production. However, an automated system that produces personalized medicine on an industrial scale is crucial to establish individual treatment at a reasonable price. Chapline et al. [30] introduce an SO system consisting of several production stations that are connected by conveyor belts as illustrated by Figure 2.5 that can establish such an industrial batch size one production for individualized medicine. There are three kinds of stations within the system:

1. Stations that load pill containers on a conveyor belt,
2. stations that dispense different ingredients, and
3. stations that remove processed containers and palletize them.

These stations are communicating with each other to self-organize the overall production process. In particular, before forwarding pill containers, the stations signal each other that they are ready to send or receive pill containers to avoid congestions on the conveyor belts. There are three different types of ingredients for the pills that are depicted in different shades of gray in Figure 2.5. Recipes specify how the pill containers should be filled through ordered applications of specific ingredient types and amounts.

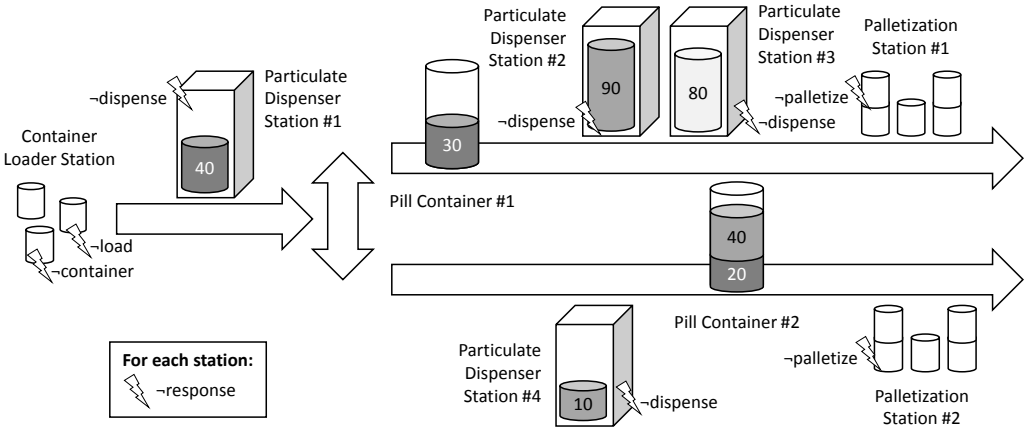


Figure 2.5. A schematic overview, taken from [71, 88], of the self-organizing production system case study for personalized medicine. Conveyor belts connect the stations; they either load pill containers onto the belts, palletize them, or dispense specific amounts of one or more ingredients into the containers. The overview shows a very simple configuration of the production cell: The pill containers can take two different routes through the system, with only the top route being able to dispense all three ingredient types into the containers. The routes the pill containers take therefore depends on their recipes and the number of ingredients left. For example, the lower route has only 10 units of the medium gray ingredient left, causing all pill containers requiring more than 10 units to be routed to the upper section of the production cell.

These recipes are unknown at design time of the system as they enter the system during run time, for instance via web services whenever a personalized product is ordered. New tasks, therefore, enter the system with a higher frequency than in the production cell case study. For each recipe, the system's SO mechanism finds a sequence of neighboring stations that are capable of processing pill containers according to the recipe. In contrast to the production cell case study, the conveyor belts establish the resource flow statically, i.e., conveyor belts cannot be dynamically rerouted similar to the carts. Besides reorganizing the system due to a new task, several environmental influences might cause a need for a reorganization to keep the system able to fulfill its goals. In Figure 2.5 there are different environmental changes shown mark with a flash. For instance, the dispenser station # 1 is no longer able to dispense pill containers into the system. In that case, a possible reconfiguration might be that dispenser station # 2 takes over this responsibility. This case is also an example showing the discrete properties of that system: Different properties are either available or not, the state changes discretely from one to another, there is no slight change.

This case study is a further system from the class of resource flow systems: the pill container is the resource establishing a flow through the system and being filled by the stations. The pill production system, as presented here, maps to the meta-model for resource system, as shown in Figure 2.4, as follows: The stations and the conveyer belt are Agents which are monitored by the Observer/Controller. The conveyer belt transports a pill container, i.e., Resources, in one direction. That is different from the production cell. Here the Resources are able to move bidirectional, where here the Resources are only unidirectional; the input/output relations of the Agents are consequently different. A Task,

which is a recipe, is requiring different Capabilities, i.e., the pill ingredients. Therefore, the conveyer belt and the stations are responsible for processing the pill container in a defined order, the Role determines that by holding the Capabilities to be applied and the route where the pill container is transported. Again, the Observer/Controller encompass the SO mechanism of the system.

Challenges for Testing

As the production cell and the pill production are applications of the same meta-model, the challenges for testing are quite similar. However, the pill production has some unique properties that need to be considered: The resource flow in this system is different, as it is unidirectional, that leads to a situation where errors might have a long propagation chain to the actual failures. A wrong routing at the beginning, caused by an erroneous allocation of the routes, might cause no direct failure if the first capabilities are applied without a problem and only at the end it might be noticed that the route is wrong. This is because the dispenser stations are highly redundant in their capabilities. Further, the tasks in this system are changing very frequently, that is called a batch size one production, compared to the production cell. That has to be respected and causes a different view on the tasks during testing. A higher variation of tasks is needed as test inputs. These two (major) differences cause the error masking, which is different, and the properties of the vast state space, needing a different treatment.

Further, the tight integration of the SO mechanisms, via digital twins, is still challenging for the isolation and, indeed, self-healing is always a concern for masking errors. Judging over the results by the test oracle might even need more insights into the complex routing problem to be solved and detect errors earlier.

Implementation Details

The concepts of this case study have been developed in the project Evolvable Assembly Systems and published in [30]. We used the descriptions and explanations as well as discussions with the authors to implement the case study as an instance of the described meta-model for SO resource-flow systems. The result is a simulation written in C# using a simple step-wise execution model that is iterating in a defined sequence over each component in each step to perform its action.

2.2.3 Self-Adaptive Webservice System: ZNN.com

The Znn.com case study is a widely established standard case study for self-adaptive systems.[8] The case study has been first described by Cheng et al. [38]. The Znn.com cases study is an online service serving different kinds of news content to its customers, like, cnn.com or sz.de, that aims at being highly reliable and responsive. Depending on different conditions the demand for this web services is highly volatile. For instance, a football game at the world championship leads to dramatical high demand on sports

[8] Its description and concepts are provided by `http://self-adaptive.org`—the website for Software Engineering for Self-Adaptive Systems, that is a community platform for numerously Dagstuhl Seminars as well as the Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS).

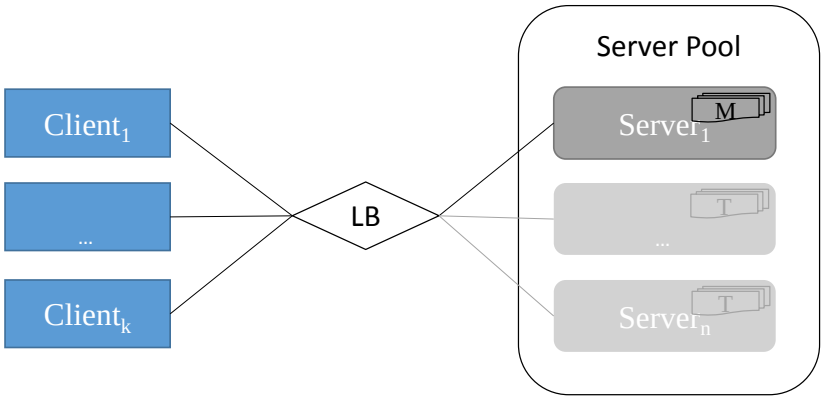


Figure 2.6. The architecture of the *ZNN.com* system consists of a number of k clients and n servers in a server pool. The clients send requests to the load balancer (*LB*) that distributes the requests to the servers and is able to switch between the modes *multimedia* (*M*) and *text* (*T*) for the servers as well as increment (resp. decrement) the server pool size.

news after the game or at half-time. Whereas that situation is rather easy to foresee other events, like a terror attack, leads to an unpredictable demand for up-to-date news on that event. Architecturally, ZNN.com is a client-server system with a multi-tier architecture model., as shown in Figure 2.6. However, the actual demand on resources is not fully predictable and assignable at design time. ZNN.com uses a load balancer to equilibrate requests across a pool of servers, the size of which is dynamically adjusted. The business objectives at ZNN.com are to serve news content to its customers within a reasonable response time range while keeping the costs of the server pool within its operating budget. From time to time ZNN.com experiences spikes in the news requests that it cannot serve adequately, even at maximum pool size. To prevent unacceptable latencies, ZNN.com opts to serve minimalist textual content during such peak times instead of providing its customers no service. The adaptation decision, for self-organizing the server allocation, is determined by observations of overall average response time versus server load. Specifically, four adaptations are possible, and the choice depends not only on the conditions of the system but also on business objectives:

1. Switch the server content mode from multimedia to textual and
2. vice versa,
3. increase the server pool size, and
4. decrease the server pool size.

Within the ZNN.com case study, the SO allows automation of adaptations that strikes a balance between multiple objectives at run time. The adaptations of the system are discrete, leading to a classification of a discrete SO mechanism.

Challenges for Testing

Compared to the previous case studies it is rather obvious how to extract the SO mechanism since it is located in one component. Nevertheless, the different possible

scenarios and situations, as well as the various configurations for the system, still lead to a vast state space that as to be addressed. That also leads to a challenge for the oracle: The different possible states of the system highly depend on the environment of the SO mechanism. It is not possible to foresee every state at design time and to judge over its correctness. Further, it is challenging to respect the interdependencies between the test inputs different. Previous test inputs influence later ones since the system adapts to patterns of the input. Such a pattern is the demand of time. The SO mechanism is using this pattern to foresee future changes in the demand and thus to be able to act proactively.

Implementation Details

The ZNN.com case study has been introduced by Cheng et al. [38] and is now used as a standard case study in the community of Software Engineering for Self-Adaptive and Self-Managing Systems. We used the descriptions provided by Cheng et al. [38] to build a simulation of the ZNN.com in a simulation environment, written in C#. The simulation uses a step-wise simulation model, where every component is performing its action in a predefined sequence.

Summary and Outlook. We introduced, classified, and discussed five different SOASs in this chapter. The used SO mechanisms have classified each case study as either discrete or continuous.

The case studies introduced in this chapter and used throughout the thesis are implemented systems from different developers, institutes, and industry. They have been selected to demonstrate the power of the testing approach presented in this thesis as well as for illustrating the concepts and methods explained in the remainder of the thesis. For each system, we discussed the particular challenges for testing resulting from the particular configuration and characteristic of the SO mechanisms that run in the case studies. The challenges are summarized as follows:

1. Making SO mechanisms testable as well as isolating and integrating SO mechanisms for testing
2. Coping with error masking of SO mechanisms in testing
3. Providing a test oracle for SO mechanisms
4. Coping with a huge state space

In the following chapters, we will show how to cope with these challenges by providing a test oracle in Chapter 3, making SO mechanisms testable in Chapter 4, isolating and integrating SO mechanisms in Chapter 5, and providing test concepts in Chapters 6 and 7.

Summary. The specification of a system is the foundation for testing the software as it defines the obligations of the System under Test (SuT). In this chapter, the concept of the Corridor of Correct Behavior (CCB) is introduced, which was proposed by Güdemann et al. [70], that enables a thorough specification of adaptation as well as SO mechanisms. We will show how these concepts are used in testing and how the constraints, forming the CCB, are unambiguous, complete, consistent, and traceable derived by the goal-oriented approach called Knowledge Acquisition in Automated Specification (KAOS). Further, the resulting obligations for the SuT are transformed, by an approach presented, into test monitors that act as the test oracle. The content and contributions of this chapter are published in [47, 154].

3

Specification of Functional Behavior of Self-Organization Mechanisms and Derivation of an Automated Test Oracle

3.1 Related Work **27**
3.1.1 Specification of SOAS 27
3.1.2 Deriving Automated Test Oracles 28
3.1.3 Runtime Verification 29
3.2 The Corridor of Correct Behavior—Specification of Self-Organizing Behavior **30**
3.2.1 The Restore Invariant Approach—Descirbing the Corridor of Correct Behavior 30
3.2.2 Application of the Restore Invariant Approach (RIA) to Software Testing 31
3.3 Goal-oriented Modeling of Functional Behavior with KAOS **32**
3.3.1 The KAOS Methodology 32
3.3.2 RELAX Goals for Introducing SO as Adaptation 33
3.4 Deriving the Test Oracle from the KAOS Model **35**
3.4.1 Process for Generating an Automated Test Oracle 36
3.4.2 Implementation of Transforming Requirement and Constraints to a Monitor Model 37
3.4.3 Implementation of the Transformation for the Monitor Model to an Oracle 39

The prerequisite for testing software is a specification of the System under Test (SuT). The specification describes the functional and non-functional obligations of the SuT. The IEEE standard 830-1984 [78] describes the main properties of a specification with unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable during the operation and maintenance phase. On the basis of that specification, testing is able to reveal possible situations where the SuT does not fulfill these obligations by executing the SuT. The situation where the specified behavior is not corresponding to the actual behavior is called a failure [111]. This failure is caused by an error, that is a state in the SuT which might lead to failure [111]. An error is caused by a fault [111], e.g., a human programming fault.

Self-Organization (SO) mechanisms, however, are built up on the premise, that decisions are shifted from the design time into the run time, which enables giving a complete specification. Because the term complete means, that the specification defines *"the responses of the software to all realizable classes of input data in all realizable classes of situations"* [78]. As a consequence the specification of SO mechanisms is underspecified according to its completeness by intention. The intention is to allow for adaptation and for SO by creating degrees of freedom, that enable decision making at run time. Indeed, a Self-Organizing, Adaptive System (SOAS) still has to operate within borders and fulfill functional as well as non-functional goals in order to be valuable and applicable. These borders and goals are forming the specification of the adaptive and SO behavior of the system.

This chapter presents an approach, that enables the definition of unambiguous, complete, verifiable, consistent, modifiable, traceable, and usable requirements for SO mechanisms, respecting the special completeness of requirements for SO mechanisms. For this purpose, we will combine different well-established approaches and extend them toward a thorough procedure, that is designed to directly generate test oracles from the specification. This minimizes human faults in testing and maximizes the efficiency in the testing process when it comes to changing requirements. It establishes a foundation for the testing methods presented in later chapters.

First the goals and borders of the SO mechanisms have to be formalized. For this purpose, Güdemann et al. [70] introduced the concept of the Corridor of Correct Behavior (CCB), that formalizes the goals and borders of SO mechanisms by constraints. Defining the CCB demands for breaking down the overall goals and specification of the system to constraints, that are defined on component level. The Knowledge Acquisition in Automated Specification (KAOS) methodology [164] provides an established framework for specifying system goals and deriving sub-goals that can be broken down to requirements. The goal-oriented specification of functional behavior of adaptive as well as SO mechanisms enables a clear traceability between high system-level goals and low-level requirements. Different levels of formalization are possible for defining goals and requirements, supporting an unambiguous and consistent definition by clearly defined relations among the goals. In general, between goals there might be an *and*-relationship to a set of sub-goals or requirements or alternatively an *or*-relationship, describing the relations within the KAOS-tree. Each goal as well as each requirement encompasses a textual description. For deriving the CCB, we use *OCL* as a description language of the requirements. The description is based on a domain model of the SuT, providing a consistency check of the specification. These requirements are traceable in the whole document and are also used for generating test oracles for SO mechanisms. The generation of the test oracle is performed in a Model-Driven Design (MDD) fashion. Modifications in the requirements document are consequently directly transformed into modifications of the test oracle.

3.1 Related Work

Specifying and verifying the output of an SuT are closely related in the area of software testing. The specification is delivering the obligations that are to be verified by checking whether or not the SuT shows the expected behavior. For software testing this verification is done by executing the system and comparing its actual output with the expected, resp. specified, output. The comparison is based on a so-called test oracle. As Barr et al. [13] or Binder [17], amongst others, state, the provision of such a test oracle is far from obvious, this is why it is often referred to as the oracle problem. Barr et al. [13] outline the necessity of having an automated oracle in software testing, but also its challenges. The need for an automated oracle is mostly argued by the following: First, an automated test process builds upon automated oracles. Second, the quality of the oracle is crucial for the quality of the performed software testing. False positives or false negatives must be prevented for a valuable quality assurance. Automated oracles are known to be less failure prone, since human error can be minimized [13]. The specification is the baseline for all approaches toward an automated oracle. As SOAS are shifting decisions from design time into run time the specification of these systems is demanding. This is of special concern for the SO mechanisms, that are responsible for the run time behavior of the SOAS. The research community follows different directions for the specification of SOAS, that we will examine. Afterward, we will take a closer look at the implementation of automated oracles for the categories of specified, derived, and implicit test oracles. These three categories are, according to the survey carried out by Barr et al. [13], the main approaches. A related direction to automated test oracles is Runtime Verification (RV). The approach presented in this chapter is related to these approaches as well, thus, we take a closer look at the state of the art of RV, too.

3.1.1 Specification of SOAS

We build upon the concepts of Güdemann et al. [70] for the specification of SOAS. Güdemann et al. [70] uses the concept of the Restore Invariant Approach (RIA) to describe the SO behavior of a SOAS. The approach is based on a specified invariant of the SOAS. In contrast to a classically known invariant it is possible, that it is not fulfilled during the system's execution, but has to be restored in order to proceed. This reflects the characteristics of SO mechanisms, that are responsible for reconfiguring, resp. reorganizing, the system in order to fulfill its duties. That is formalized by the RIA and can be described as an CCB for the SOAS. Thus, by having the invariant the system is either inside the CCB, with no need for SO (only for optimization purposes), or outside the CCB, if the invariant does not hold and a reconfiguration is needed. Schmeck et al. [143] introduced a similar concept with the three spaces of SOAS: the acceptance, survival, and dead space. The acceptance space corresponds to a holding invariant, the survival space to a violated invariant that can be restored, and the dead space to an invariant that does not hold and cannot be restored. The invariant of the RIA has to be specified by a set of conjunct constraints for the system. The derivation of these constraints from the system goals, thus establishing traceability, is an extension of the concepts presented by Güdemann et al. [70] as well as the linking of the constraints with a domain model, making the model consistent and unambiguous. This is needed

for using the approach for deriving a test oracle and having a specification complying with the standards (cf. [78]). Cheng et al. [36] as well as Whittle et al. [171] are also investigating the specification in a traceable, consistent, and unambiguous fashion. These developed concepts are suited for adaptation, we will extend these concepts to SO in this chapter. This is done by combining the goal-oriented approach with the concepts of the RIA. Cheng et al. [36] showed how the concepts of KAOS, originally introduced by van Lamsweerde et al. [164], are applicable for adaptation of system parameters at run time. We will show how these concepts can be extended to SO behavior and gainfully integrated with the RIA. A different direction for specifying adaptation was presented by Morandini et al. [105]. The specification is based on alternative configurations for the system. These configurations are available for adapting the system in a particular situation. Thus, the scope of actions is limited at design time.

3.1.2 Deriving Automated Test Oracles

A system's specification delivers obligations to be evaluated for a test oracle in order to judge over the test result. A specification of a SOAS is different compared to classical system due to its underspecification. But, given the RIA, we are able to clearly specify the responsibilities and thus are able to distinguish a failure state from a correct state. Thus, techniques for deriving automated test oracles are indeed related to the approach proposed in this thesis, but not sufficiently applicable to SO without the concepts described in this chapter.

In general, there are three different categories of automated test oracles, according to Barr et al. [13]: the specified, the derived, and the implicit test oracle. The approach presented in this chapter is classified as a specified oracle, where the specification is made in model-based fashion. A model-based approach uses some kind of model to specify the expected behavior of the SuT. Utting et al. [163] show a variety of different approaches. In most cases, the behavior is described via Unified Modeling Language (UML) state chart or sequence diagrams, specifying valid states and state changes, or resp. interaction sequences. The ability of the model to abstract and condense is used to keep the models and the oracle manageable. The modeling approach used in this chapter is based on the UML. However, by using the concepts of the CCB for specification, it differs from the classical model-based approaches. These are specifying all acceptable state-transitions for SuT, that is not possible for an Self-Organization Mechanism under Test (SOuT). The usage of assertions and contracts (also constraints) is a further specification-based approach for generation of the test oracle. Hoare [76] introduced the concepts of assertion and contracts. The approach enables to write the intention of the program into the code with the ability to check the code. A prominent development that emerges from Hoare's approach [76] was the Eiffel programming language [100]. In this way, the expectations to the system were stated explicitly. A different approach for the test oracle is the so-called metamorphic approach, which is a derived test oracle. For this purpose, a relation between the input and the output is defined. Chen et al. [32, 33] introduced that concept for testing software where it is difficult to specify the expected output for each and every input. Using the CCB follows this idea and concept, since it also does not specify each and every combination

of input and expected output. However, the CCB is not a defined relationship that is described for an input. Still, the CCB can be seen as a derived concept as well as an implicit one. Implicit, because the CCB and its constraints are not directly describing the intended behavior, but the behavior to be restored by an SO mechanism. This is similar to concepts of implicit test oracles which are classically detecting anomalies like abnormal termination caused by a crash or an execution failure, as proposed by Cadar et al. [24]. Thus, the approach for a test oracle for SO mechanisms, that is presented in this chapter, combines different concepts and different approaches for deriving test oracles in order to handle the challenges of defining an oracle for SO mechanisms.

3.1.3 Runtime Verification

The approaches of automated test oracles are similar to approaches of RV, as proposed by Leucker & Schallhart [90]. RV is concerned with monitoring a particular part of the system, during its execution with the intention of checking a proposition. Here, the verification process of the proposition is shifted from design time into run time. The reasons for this shift are argued quite similar to the shift made for SOAS: the complexity of the system does not allow for proving the correctness of an implementation according to an obligation or going over every possible state, thus, it is done at run time for the current state. Due to this direct relation of SOAS the concepts are also related and particularly applicable for monitoring SO mechanisms. Rosu et al. [138, 139] presented an approach for RV of safety properties of a system. Therefore, they designed a procedure, that is able to generate monitors, for RV, from Linear Temporal Logic (LTL) formulas. However, the generation algorithms are highly specialized for some particular safety properties and need customization for the concrete system. The approach is similar to the work by Goodloe et al. [67], which has emphasis on real-time safety properties. Both are different compared to the approach presented here for generating the test oracle from the requirements in an MDD fashion, since they are focused on different system properties and types and are not directly applicable as a test oracle for SO mechanisms. Jin et al. [81] follows a generic approach for generating RV monitors by developing an Domain Specific Language (DSL) for the monitoring approach by Manna et al. [95, 96]. The DSL is used for generating a monitor for locks in Java programs. However, there is still no traceability and generality as with the MDD approach presented in this chapter. The approach by Demuth et al. [44] is more general by using the Object Constraint Language (OCL) as an input language. The target language for checking is Java, which is not replaceable. Further, all approaches so far are not concerned about the special characteristics of SOAS. Calinescu et al. [25] have developed an approach for adaptive systems. Here, the monitoring is based on a global model of the system, that is updated with the current state of the system and is model checked in a quantitative approach. The approach of this chapter is more focused on qualitative checking, as common for software testing. Further, we have also the ability to use requirements for SO mechanisms. The difference is, that here more than one component of the SOAS is often involved. The needed extension is made in this chapter on the basis of the RIA.

3.2 The Corridor of Correct Behavior—Specification of Self-Organizing Behavior

In contrast to classical systems that have a complete specified behavior of each input and related output, SOAS are known to be underspecified. This underspecification is by intention. The underspecification is enabling autonomous adaptation of the system's parameter, settings, or organizational structure, amongst others. Indeed, not each and every non-SO system is complete in its description in a narrow sense, since the effort of providing this complete definition of the system is too large. This is often mitigated by defining the system's obligation in less precise or abstract notation, e.g., structured text. Nevertheless, the aim of the specification is to state the complete list of responses of the system. Checking these responses by executing the system with the intention of revealing deviations between the actual and the intended behavior is called software testing. The same holds for testing SOAS, but the definition of the system's reaction to all possible inputs is incomplete. In order to adequately test, and foremost judge over the result of the test cases, a specification is needed that allows for evaluating system states as correct or incorrect. The CCB describes the needed obligations of self-organizing as well as adaptive behavior based on the behavior of the controlled system. This is done by forming a corridor in which the behavior of the system is correct, i.e., no need for adaptation nor for SO. The corridor is formed by simple constraints, that describe invalid states of the system and consequently the need for the adaptation or SO mechanisms. Consequently, the CCB allows for defining the functional behavior of SOAS and is the foundation of the test oracle for SO mechansims.

3.2.1 The Restore Invariant Approach—Descirbing the Corridor of Correct Behavior

Güdemann et al. [70] introduced in the RIA the CCB for SOAS. RIA's concept is to specify a set of conjuncted constraints that should be fulfilled by the system while it is executed. If a constraint is not fulfilled in any point in time the system is able to restore its invariant, i.e., reconfigure the system so that the conjunction of all specified constraints holds, by adaptation or SO mechanisms. Figure 3.1 illustrates this responsibility and its obligations by showing the CCB of the system. The CCB is formed by the conjuncted constraints, i.e., the INV_{RIA}. Inside the corridor all constraints are satisfied by the system, i.e., INV_{RIA} holds, whereas outside the corridor at least one constraint is broken, implying $\neg\text{INV}_{\text{RIA}}$. Whether or not a system is inside the CCB or not is evaluated at each system's state.

Definition of the RIA

The RIA [70, 109, 110] is defined on a labeled transition system for the system $SYS = (S, \rightarrow, I, AP, L)$, with S a set of states, $\rightarrow \subseteq S \times S$ a transition relation, $I \subseteq S$ a set of initial states, AP a set of atomic propositions and $L : S \rightarrow 2^{AP}$ a labelling function. Within such a system, a trace π is a sequence of states $\sigma_i \in S$, related via $\rightarrow$ and starting from an initial state $\sigma_0 \in I$. For each state S it is possible to evaluate whether or not it is inside the corridor, like s_1 in Figure 3.1, or outside the corridor, like s_{vio} in Figure 3.1, by evaluating INV_{RIA}. The evaluation of INV_{RIA} can be performed based on the constraints

$\phi \in \Phi$ forming INV_{RIA} and having the following relation:

$$(3.1)\ \forall s_i \in S : \bigwedge_{\phi \in \Phi} (\phi(s_i)) \rightarrow \text{INV}_{\text{RIA}}$$

Each constraint $\phi \in \Phi$ is defined within a given context. This context is a component of the system SYS which is constrained by ϕ. The RIA, further, allows for a violation of INV_{RIA}, as shown in Figure 3.1 with s_{vio}, if INV_{RIA} is restored in the next step. Thus, RIA does not demand for $\Box\text{INV}_{\text{RIA}}$, but for $\Box\text{INV}_{\text{RIA}} \vee \Box(\neg\text{INV}_{\text{RIA}} \rightarrow \circ\text{INV}_{\text{RIA}})$ In a classical system, a violation of the specification is not allowed and is denoted as a failure when it is observed. For SOAS, the obligation is weaker and further specified less detailed; the CCB only specifies what is not allowed, but not a concrete (complete) specification of expected behavior for a given input.

Implications of the RIA

Equation (3.1) further indicates, that if there exists a constraint $\phi \in \Phi$ where $\neg\phi(\sigma_i)$ holds, then $\neg\text{INV}_{\text{RIA}}$ holds for the whole system. This follows from the conjunction of the constraints of the system, i.e., each constraint must be fulfilled. Having this relation, it is possible to monitor the constraints $\phi \in \Phi$ separately and further to observe them locally. This is exploited by SO mechanisms that are working in a decentralized fashion. Further, we are going to exploit it by monitoring the constraints $\phi \in \Phi$ separately by decentralized test oracles.

3.2.2 Application of the RIA to Software Testing

For software testing the CCB can be used as an oracle, to decide whether an adaptation or an SO mechanism performs as specified. Figure 3.1 illustrates the specification of the CCB by marking the correct (checkmark) reconfiguration as well as an incorrect (cross) behavior, based on a system state of the system controlled by the corresponding adaptation or SO mechanisms. In general, if a constraint is violated it has to be recognized and processed in a way that the next system state restores the INV_{RIA}. Despite the fact that the system is, in classical sense, underspecified according to an enumeration of correct and incorrect state-transitions, the CCB provides a complete specification as needed for test specification for autonomous behavior of adaptive and SO mechanisms.

This can be illustrated by a small example based on the self-organizing production cell. Let us assume the INV_{RIA} consists of only the following constraint:

context Agent **inv** capabilityConsistency:
self.availableCapabilities
→ **includesAll**(self.allocatedRoles.capabilitiesToApply)

The constraint says, that only available tools (here in general capabilities) are allowed to be selected by an agent (e.g., a robot). In case a robot has the tools for drilling a hole and inserting a screw it may only be assigned to apply these tools. The adaptation task in this case is triggered by the loss of a capability, e.g. the drill breaks at some point in time. Classically, one would expect a specification that specifies each condition and situation where a concrete tool that is known at specification time should be applied. Indeed, in

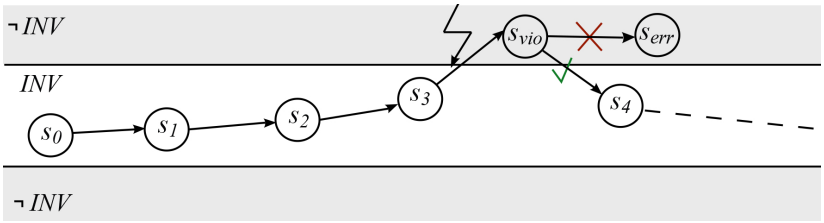


Figure 3.1. A transition system denoting particular states and their transitions within the configuration space of the SOAS, according to Nafz et al. [**?**]. The states $s_0, \ldots, s_4$ are shown inside the corridor, i.e., the INV_{RIA} holds implying that all constraints specified for the system are fulfilled. If a state is in the region of $\neg\text{INV}_{\text{RIA}}$ (i.e., at least one constraint does not hold), this violation has to be recognized and fixed by an adaptation or SO mechanism. If the transition from s_{vio} goes to a state outside the corridor, as for s_{err}, this is a failure of the SO mechanism or the adaptation mechanism that controlled the system. A correct action of the mechanism is to transfer the system state back into the corridor, as for s_4, if possible.

this case also the loss of a capability can be compensated. However, the specification is bound to the knowledge available at design time, e.g., the concrete products that are assembled, the concrete tools that are available, the concrete actions to be taken if one tool is missing and another robot (also a concrete one) needs to take over the task. Having specified the CCB in contrast enables to have unknown tools, unknown tasks, and consequently unknown situations to adapt to. The underspecification enables this. Nevertheless, it is still fully testable by the CCB. The adaptation mechanism needs to recognize if a capability is selected as a role to apply which is no longer available and needs to take appropriate actions that this situation is not present afterward. Thus, the CCB makes adaptation and SO specifiable for testing. The quality depends, however, on the quality of the definition on the CCB.

3.3 Goal-oriented Modeling of Functional Behavior with KAOS

Specifying the functional behavior of SO mechanisms is supported by the CCB. The obligations are therefore described by constraints for single system components. Defining these constraints adequately is a challenging task. The quality of the constraints are determining the quality of the SOAS and testing, as it is described in this thesis. In order to support this process, we follow the goal-oriented modeling paradigm and extend it toward modeling the CCB with it.

3.3.1 The KAOS Methodology

The goal-oriented modeling approach we use is called KAOS, introduced by von Lamsweerde et al. [164]. KAOS allows for a traceable, consistent, and unambiguous specification of requirements in the form of goals. For this purpose, a lean concept for specifying the system goal and putting it into relation with sub-goals and derived requirements is supplied. Basically the KAOS methodology consists of a graphical and a textual formal description of the desired system. The graphical notation describes as well as connects the goals, requirements, agents and obstacles. The connections specify refinements,

assignments, or obstructions. These elements are used to model a goal refinement graph with specific system requirements at its leafs (cf. Figure 3.3). To build this graph, we are using only a subset of the methodology, i.e., the refinement consists only of logical *and*s, implying that all requirements have to be achieved to fulfill the global system goal. This is a consequence of applying the RIA where the INV_{RIA} is a conjunction of constraints, i.e., requirements in the KAOS speak. Figure 3.3 is showing a simplified KAOS model of the energy grid case study. The graphical elements shown are representing system goals and requirements, whereas the requirements are forming the leafs of the goal tree. The goals are refined by the edges and assigned by a different form of edges to so-called agents. These agents are responsible units or components in the system to be developed.

3.3.2 RELAX Goals for Introducing SO as Adaptation

The KAOS methodology is not tailored for adaptive nor self-organizing systems. The aim is still to form a complete set of requirements within the goal model. That is, as discussed before, not intended for SOAS. The main difference, taken into account by the RIA is the fact of uncertainty at design time leading to an underspecified set of requirements in the classical sense: RIA only specifies the correct and incorrect states, but not the allowed transitions between the states, i.e., the actions to be taken. The intended use of the KAOS approach is to specify the CCB by deriving all constraints, in form of requirements, from the system goals. This allows for tracing the constraints as well as having a consistent model and unambiguous set of obligations. For the latter, the use of a conceptual domain model of the system as well as the formulation of consistent OCL constraints is used.

Cheng et al. [36] propose an extension of the KAOS methodology that allows to express requirements for adaptive systems by incorporating uncertainty factors the system is supposed to adapt to. These uncertainties are identified with the help of the conceptual domain model. Their existence can lead to a reformulation of requirements or introduction of new requirements, effectively introducing requirements for system adaptivity. The proposed process, shown in Figure 3.2, is structured into four parts, which are performed iteratively. Since it uses progressive refinements from system goals to individual requirements of the agents, it can be easily integrated in an iterative-incremental software engineering process. The used steps are:

Identify Top-Level Goal: The refinement process starts with a global goal for the intended system. Based on this global goal the top-level goals are derived which are necessary to fulfill the global goal.
Derive the Goal Model: These top-level goals are the basis for the complete goal model, which is developed by refining the goals until clear and achievable requirements can be formulated for the fulfillment of the goals.
Identify Uncertainty Factors: This goal model has to be examined to identify uncertainty factors, i.e., obstacles that might prevent the system from reaching the goals.
Mitigate Uncertainty Factors: These obstacles can be mitigated by reformulating the goal model, introducing new goals, or changing existing goals.

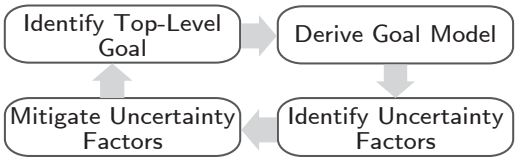


Figure 3.2. The requirements engineering process, according to [36], starting with the identification of the top-level goals of the system, which are refined in the next step. The resulting goal model is afterwards examined according to find uncertainty factors. This can be mitigated, e.g., by introducing new goals. This process is iteratively executed until all uncertainties are eliminated.

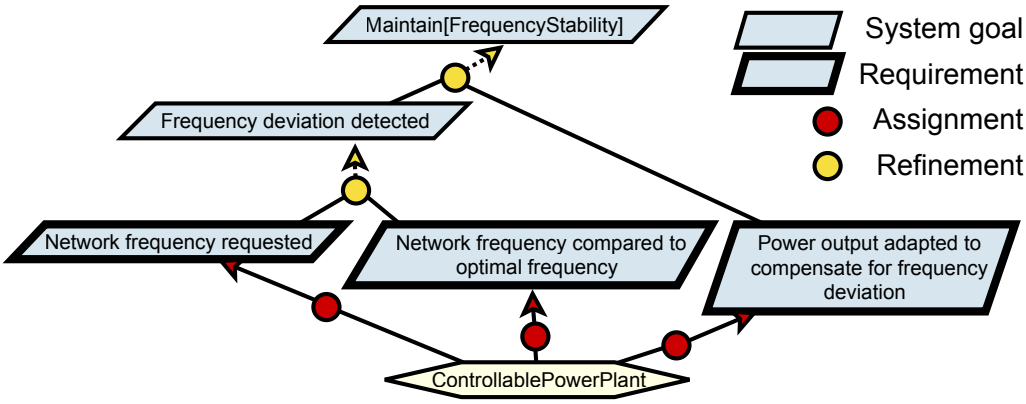


Figure 3.3. The goal refinement graph in KAOS' graphical notation for the goal "Maintain[FrequencyStability]".

After mitigating the uncertainties, the process has to be repeated, until a sufficient model is developed. As part of this refinement process, the requirements need to be augumented with a formal specification of the system goals and constraints in OCL.

The described process by Cheng et al. [36] is not necessarily needed to have a well-defined CCB, however, it turned out be very useful for its development.

Let's consider a small example from the energy grid case study. We use here the approach by Cheng et al. [36] to develop a constraint for the CCB. In autonomous power grids, scheduling of controllable power sources is performed based on predictions of output and demand. These predictions are based on a number of uncertain factors. Therefore, even the best scheduling algorithms will never be able to approximate the required demand and the so-called "residual load", i.e., the power that needs to be produced when all production by non-controllable power plants (solar, wind, residential heat-and-power) has been factored in. However, the power grid is very sensitive to deviations between production and demand and therefore, there needs to be an adaptive mechanism that can quickly react to such deviations. Since deviations alter the power grid's internal frequency, all power plants can monitor this frequency and react to deviations from the optimal frequency autonomously.

The necessity of adapting the power plants' output based on the network frequency is captured in the goal "Maintain[FrequencyStability]". It can be refined to concrete requirements for controllable power plants as shown in Figure 3.3. They need to measure

the frequency and compare it to the optimal frequency, reacting to deviations by adapting their output. These requirements capture the control loop: changing the output has a direct effect on the network frequency, thus providing feedback. The constraint that needs to be observed corresponds to the requirement "Network frequency compared to optimal frequency". In OCL it can be expressed as:

```
context ControllablePowerPlant inv noFrequencyDeviations:
 (currFrequency - optimalFrequency).abs() < allowedDeviation
```

While this constraint may seem simplistic, it is a good example for a property that has to be monitored as part of a feedback loop in an adaptive system.

The monitoring and the controlling part is both due to the adaptation and SO mechanisms. Thus, the test obligations can be directly derived from the constraints that form the CCB. The derivation can further be automtomated due to the tooling used for modeling the KAOS model in Objective [137] that is supplying a processable *xmi* format.

3.4 Deriving the Test Oracle from the KAOS Model

The oracle problem is a well-known challenge for all testing endeavors [13, 17]. However, the properties of SO mechanisms increase this problem: for classical testing the conditions of execution for the SuT as well as the concrete requirements are known. Let us call these facts the "known-knowns". For SOuT we know that there are unknown conditions of execution where we can hardly decide a priori, i.e., at design time, whether a state is correct or not; we call these conditions the "known-unknowns". Moreover, for the SOuT there might even be situations we are not aware of at all, which we call the "unknown-unknowns". An oracle that is capable of evaluating the test results of SO mechanisms at least has to be able to handle the "known-unknowns".

For an instance of "known-unknowns" consider a smart energy grid setting where different power plants are self-organized in different so-called autonomous virtual power plants. If weather-dependent power plants are included, the SO mechanisms here will depend on the weather conditions. We know that there are different conditions like sunny, rainy, and windy, but we also know that we do not know all different possible combinations and the according correct organizational structures at design time of the test (or at least we cannot compute all). However, an oracle has to cope with that and has to decide whether a result is accepted as correct or rejected as incorrect. The CCB forms the test obligation for adaptation as well as SO mechanisms. Having derived the complete KAOS model, we now have a traceable, consistent, and unambiguous definition of the obligations to be tested. For testing we are able to use this information as the foundation of the test oracle. There are two properties of the CCB, which we can exploit for this purpose: (1) the constraints used for all different kinds of SOAS have shown to be rather simplistic and straight forward to evaluate, since they have been broken down to single components and (2) all constraints of the INV_{RIA} can be evaluated locally at one corresponding agent or class of agents. This simplifies the task of an automated procedure of checking whether constraint violations have been detected and treated in a way, that the system is inside the CCB afterward. To build a highly reliable oracle, the process of generating the oracle is automated, thus, we minimize the human error.

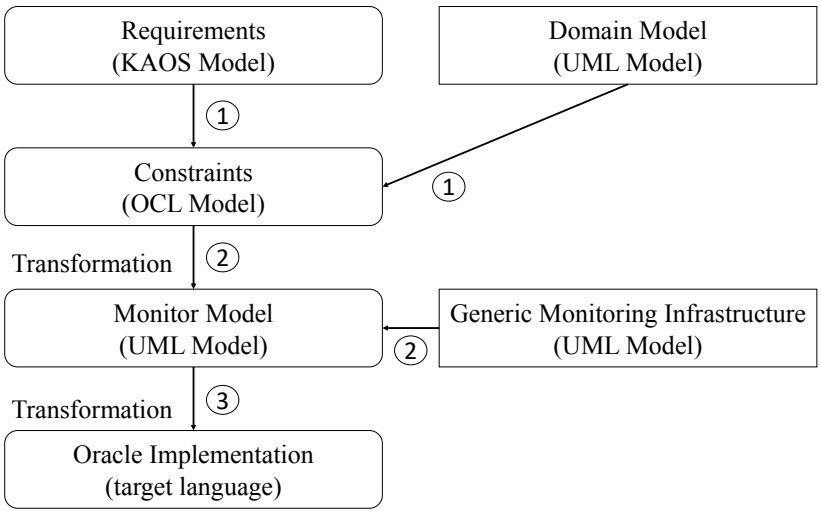


Figure 3.4. The transformation process, starting from requirements modeled in KAOS that are formally described while the requirements become clear and a domain model is elaborated, to abstract monitors expressed as UML class and activity diagrams backed by a model of the monitoring infrastructure, to the final implemented oracle for the target platform.

The KAOS model, containing the constraints—formalized as OCL constraints—, is for this purpose transformed into an infrastructure used as an automated oracle in a test setting.

The synthesis of the oracle infrastructure follows a model-driven process, outlined in Figure 3.4, and is divided into three major steps, which can easily be integrated into an iterative-incremental design process. The process can be repeated when requirements or the domain model change in an MDD approach. Changed parts of system models and implementation will be re-generated while existing models and code are preserved.

3.4.1 Process for Generating an Automated Test Oracle

The generation of the automated test oracle is divided into the following steps:

Step 1: System goals and constraints are described formally during the iterative process of requirements analysis, shown in Figure 3.2. During this process, a domain model is created that can be used to express constraints in OCL—a language arguably more accessible to system designers than the aforementioned temporal logics—that formally describe the requirements. These OCL-constraints $\phi \in \Phi$ define the correct states of the system and are therefore transformed in a model-to-model transformation to an abstract monitor model when moving from the test analysis stage to the test design stage in each iteration. The state-based evaluation of the OCL-constraints, i.e., to check for all $\phi \in \Phi$ whether $\phi(\sigma_i)$ with $\sigma_i \in S$ holds, conforms well with the semantics of the transition system the RIA is based on as described above.
Step 2: The underlying structure of the monitor model, which is created during the transformation process, contains the refined elements from the domain model and the generic monitor model. In addition, for each agent that has to be monitored, a monitor

class is created and for each OCL-constraint, a new abstract constraint class is created. Furthermore, the validity check for each constraint $\phi(\sigma_i)$ is modeled as an UML activity diagram in the constraint classes.
Step 3: The monitor model is transformed into code from which the actual distributed test oracle for the SuT can be compiled. This transformation is done when moving from the test design to the test implementation stage. This last transformation is highly specific to the target system.

If the generic monitoring model proposed here is used, the test engineer has four responsibilities left:

1. formally describe the constraints based on a domain model;
2. create platform-specific transformation rules for the target platform;
3. create code for possible additional checks by the oracle and
4. create code for the state update.

The rest of the infrastructure is created by the transformations.

The additional checks are necessary for implementation-specific properties. These are depending on the concrete implementation of the SOuT and are used to evaluate intermediate steps this is mostly helpful for fault-localization, finding the cause of the failure in order to remove it. In Chapter 6, we will show these specific properties for the case studies presented in Chapter 2.

3.4.2 Implementation of Transforming Requirement and Constraints to a Monitor Model

After the relevant requirements have been defined with OCL constraints, as described in Section 3.3, the resulting requirements model and the domain model are transformed into a monitor model, which is the basis of a specific monitor implementation. The transformation is defined in QVT (Query View Transformation) [117], which uses *Queries* on the source models to *Transform* them into target models. Part of the views can be pre-specified. We use this feature to specify a generic monitor model, as depicted in Figure 3.5, that provides the structure for the monitor models and is based on the *Observer* pattern by Gamma et al. [66].

The relevant interaction between the classes is depicted in Figure 3.7. Whenever a `MonitoredAgent` registers a change (basically, a transition in SYS from σ_i to σ_{i+1}), it informs its `Monitor` by sending the state model with the updated information. The `Monitor` then updates its state model for σ_{i+1} of the agent and evaluates all `Constraints` $\phi \in \Phi$. If one of them evaluates to false, i.e., $\neg\phi(\sigma_{i+1})$, the `Monitor` informs the executing test suite.

The requirements model and the domain model are transformed into a new monitor model, which is described by an UML class diagram and activity diagrams. These diagrams are supplemented by the generic sequence diagram in Figure 3.7 which characterizes the interaction in the class diagram. The activity diagrams, as shown in Figure 3.6, embed the OCL-constraints in the `holds()` methods of the implementations of the `Constraint` interfaces.

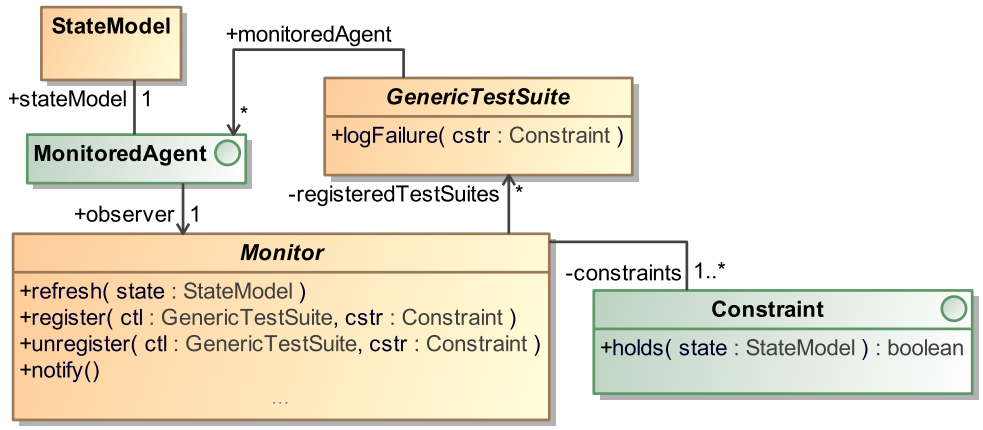


Figure 3.5. Simplified generic monitor model as used in the transformation, specified as an UML class diagram.

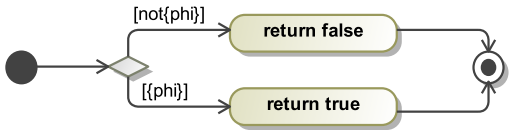


Figure 3.6. The generic UML activity diagram with the placeholder `phi` for the constraint to be checked.

Identify Observed Object: First, to generate the monitor model it is necessary to identify the classes which should be monitored. These are all classes from the domain model which have an according agent in the requirements model that has a set of semi-formally defined requirements to be monitored. Each class identified in this manner has to implement the `MonitoredAgent` interface in the monitor model.
Generate Monitoring Structure: Furthermore, a specific state model for each of these classes is created by generating a simple class containing all attributes of the agent class which represents the state of the observed class. Additionally a monitor derived from the `Monitor` class for this agent is created. Thus, a relationship between the object that should be observed and the concrete monitor is generated. Another relationship is generated between the specific monitor object and the test suite. Most importantly, the specific constraint class is added in the model.
Generate Dynamic Model: The final step in the creation of the monitor model is to parse the OCL statement and put it into the guards of the activity diagram that describes the functionality of the `holds()` method of this constraint class. The generic activity diagram for the constraint type is used by copying it and replacing the wildcard with the specific extracted guard.

This procedure is repeated for every monitored agent. After the complete transformation there is one monitor per agent, but there are several constraints per monitor. The monitors are used as an oracle in the test setting. The main duty of the oracle is to log failures, as shown in Figure 3.7. This is done by checking the system with the monitor after the reconfiguration by the SO mechanism as well as after each system step in order

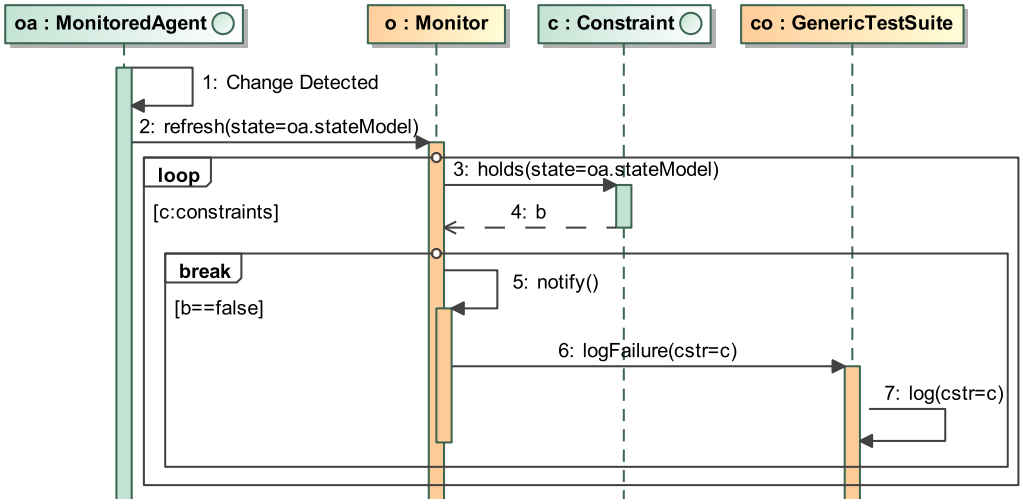


Figure 3.7. A simplified sequence diagram showing the interactions between the elements of the generic Observer/Controller model.

to check whether a necessary reconfiguration has been detected. Figure 3.8 depicts the elements that are used in this transformation process as well as a simplified version of the resulting class diagram. The transformation creates class diagrams for all agents, as well as diagrams that model the respective methods for the new constraint classes and the monitor. It also checks the domain model and the requirements model for inconsistencies, e.g., if the requirements model defines agents for which no class exists in the domain model. The result of the transformation process is a platform independent model of the monitoring infrastructure specified in the UML2 meta-model of the Eclipse Modelling Framework (EMF).

3.4.3 Implementation of the Transformation for the Monitor Model to an Oracle

The final transformation to the actual monitor (that is used as the test oracle) implementations contains many platform-specific choices, e.g., whether or not monitors are independent agents or become part of the agents defined in the domain model, whether properties of the agents can be accessed directly or only via message passing, etc. It will therefore have to be adapted to each target platform and target system. However, some of the basic principles remain the same, regardless of the transformation target.

We developed a template which can be adapted for the use in a specific target system. This template is defined in the language Xpand, which is a part of the EMF and enables model-to-text transformations. The template contains generic parts, written in the target programming language that do not change and only have to be copied into the source code files. In addition, it contains dynamic (resp. system-specific) parts that depend on the elements in the monitor model and are evaluated when the transformation is performed. This final transformation consists of two steps:

Structural Transformation: First, stubs of the classes contained in the class diagram of the monitor model are generated. As defined in the static part of the transformation template, these stubs are integrated into the target platform, e.g., the multi-agent plat-

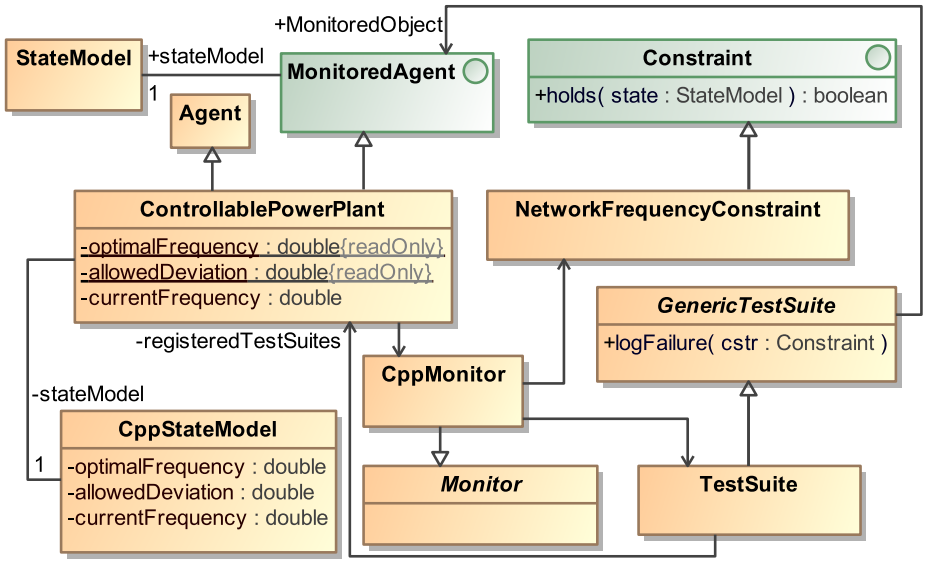


Figure 3.8. This simplified transformed model is taken from the energy grid case study. The `ControllablePowerPlant` is equipped with a monitoring infrastructure for the `NetworkFrequencyConstraint`.

form or middleware the system will run on and can contain additional initialization or bookkeeping code. The implementation of the generic sequence diagram depicted in Figure 3.7 is also part of the generic part and thus has to be included in the template. This step transforms the entire class diagram into system-specific source code.

Functional Transformation: Afterward, the activity diagrams are transformed into implementations of the `holds()` methods in the abstract specializations of `Constraint`. For this purpose, the OCL statements in the transition-guards have to be translated into conditional statements expressed in the language of the target system. The dynamic part of the template parses the OCL statement into an Abstract Syntax Tree (AST), including the constrained attributes, the OCL functions, logic operators, etc. The AST can be transformed into code for the specific target system. We use the Eclipse OCL grammar as a basis and employ a custom ANTLR[1] parser that supports all standard OCL constructs. In comparison to using a standard parser such as the one provided with Eclipse OCL[2] or Dresden OCL[3] directly, this gives us more flexibility with regard to the target language and thus the transformation target.

In concrete terms for the monitor model of the autonomous power grid, shown in Figure 3.8, the output of this last step is as follows: The target system is the middleware TEAMS [9], which is implemented in the programming language Java. So to generate the outline of the class diagram for each interface and class of the monitor model, a file containing the basic class declarations, including attributes, method stubs, etc. has to be generated. Next, the `holds()` method of the class `NetworkFrequencyConstraint` is translated into working Java code. For this purpose, the AST created by the OCL parser

[1] `http://www.antlr.org`
[2] `https://projects.eclipse.org/projects/modeling.mdt.ocl`
[3] `https://github.com/dresden-ocl`

is translated into Java syntax, e.g., an OCL "`and`" into a Java "&&" and OCL methods such as "`includesAll`" into corresponding Java constructs.

Instead of parsing the constraints, it would also be possible to use the faculties of tools like Dresden OCL, developed by Demuth et al. [44] to check the constraints at runtime. However, a parser like the one used here can be used to create code for target systems other than those based on a Java Virtual Machine which Dresden OCL is limited to as it uses Bytecode weaving. In principle, tools like Dresden OCL do not provide data gathering facilities or infrastructure for interactions with controllers. They can thus be used for constraint checking within the monitoring infrastructure, but not as a replacement for the concepts proposed here.

The classes and sequence diagrams have to be translated into the target programming language and the target platform, i.e., the multi-agent platform or middleware the system will run on. Sequence diagrams become implementations of the methods of the classes. The OCL constraints are parsed to conditional statements within `Constraint.holds()` for the corresponding constraint classes. Note that the multiple inheritance in the class diagram can be implemented differently, depending on the target language. In Java, an interface can be used, especially if the methods are abstract methods anyways.

An important decision has to be made with regard to the flow of information at this point. Changes in the `MonitoredAgents` cause updates to an internal model of the `Monitor` on which the constraints will be evaluated. Alternatively, the monitor or the implementations of the `Constraints` can request the required information from the `MonitoredAgents` directly. Which choice is best depends on many properties of the system, including whether message-based communication is used, how often the information changes and how complex an internal model would be.

A concrete monitor implementation will have to be coupled with the test suite which is executed. The transformation creates stubs for the appropriate classes and methods such as `activate()`.

Summary and Outlook. The specification of an SuT is the foundation for testing, it delivers the obligations the system has to fulfill. As SO mechanisms are characterized by making decision at run time which are actually specified at design time we called them underspecified. Indeed, the SOAS (controlled by SO mechanisms) still has to operate within borders and fulfill system goals, these borders and goals can be described by the CCB. The CCB is defined by constraining the SO and thus specifies the functional behavior of SO mechanisms and allows for the needed degrees of freedom for the SO mechanisms to autonomously make decisions at run time. In order to use the concept of the CCB, the specification needs to be traceable, consistent, and unambiguous, as defined by the IEEE standard 830-1984 [78]. The goal-oriented modeling paradigm of the KAOS approach delivers the necessary toolset which is used to break down the systems goals of the SOAS to define the constraints and form the CCB. The goal model enables to trace the single constraints back to overall goals. Further, it provides a consistent model with the coupled domain model of the system. The domain model also enables to be formulated the constraints in an unambiguous fashion by using the OCL constraint language. We further showed that the formalization of the CCB enables to derive a test oracle by transforming the KAOS model in an MDD fashion. The result is a highly reliable test oracle which is not prone to human errors. It builds a reliable foundation for the further presented testing approach and solves one of the key challenges of testing SO mechanisms: the oracle problem.

Summary. The software architecture describes the principal design decisions made concerning the components, subsystems, and relationships amongst them in a system. As a good software engineering practice, the development of an architecture is based on patterns. The here presented Corridor Enforcing Infrastructure (CEI) is an architectural pattern for SOAS, that enables testability for SO mechanisms. Further, based on the concepts of the CEI and the CCB the definition of a failure for an SO mechanisms is given, and failures are categorized. However, it is possible to show that the realization of these concepts is possible for a large system group, which can be tested on that foundation. The content and contributions of this chapter are published in [48, 52, 72].

4

A Testable Architecture for Implementing Self-Organization Mechanisms

4.1 Related Work **45**
4.2 The Corridor Enforcing Infrastructure: An Architectural Pattern for Self-Organizing, Adaptive Systems **46**
4.3 Failure Definition and Categorization of Self-Organization Mechanisms . **48**
4.3.1 Weaker Notion of Correctness for Self-Organization Mechanisms: Definition of Failure 48
4.3.2 Boundaries of Self-Organization: Tolerable and Intolerable Environmental Faults 50
4.4 Prerequisites and Benefits for Testing Based on the Corridor Enforcing Infrastructure **51**
4.4.1 Gain for Testing Based on the Corridor Enforcing Infrastructure . . . 52
4.4.2 Realizations of the Concepts of the Corridor Enforcing Infrastructure: Application Cases 52

The architecture of a software system is *"the set of principal design decisions about the system"* [157]. The design decisions made within an architecture are influencing the capabilities of the software. For instance, the design of the communication architecture of a system is determining the scalability capability; thus, a peer-to-peer infrastructure is more scalable than a central switch, whereas, the central switch is less complicated. In general, there are quite a lot of decisions to be made for setting up the architecture of a software system. Each decision influences the capabilities of the system and comes with advantages and disadvantages. It is of utmost importance to be aware of all decisions and their consequences. As a good software engineering practice, the development of a software architecture is grounded in patterns. Patterns enable to rely on the collective experience of software engineering experts in order to provide a high quality of the architecture [23]. Different quality aspects are of concern and have been considered for the architectural design. On the one hand, the functional properties of the system need to be fulfilled. The functional aspects are either directly visible to the user or influence aspects of the implementation of a function visible by the user. On the other hand, the non-functional properties are beyond a specified functionality. Typically these aspects

are reliability, compatibility, cost, maintainability, or testability [23]. We focus on the testability aspect of an architecture here: a testable architecture for Self-Organization (SO) mechanisms. Indeed, all other properties have some merits, too, and play a role in developing a successful system. However, they are not within the scope of this thesis. We will discuss the testability of architectures as well as investigate a general architectural pattern for SO mechanisms which brings this capability with it.

Testability of Software Architectures & its Challenges for Self-Organizing, Adaptive Systems The testability of an architecture describes the support of an architecture to ease the evaluation of its correctness. Testability of an architecture is according to Freedman [65] provided if it supplies the capabilities to control and observe the software. Controllability is necessary to execute a test suite and setting up the system. Observability is needed to monitor the system while the test suite is executed in order to evaluate the results. Providing these two capabilities for the architecture of SO mechanisms is far from obvious. This follows from the responsibilities of SO mechanisms of *"[modifying] the [system's] behavior and/or structure in response to [its] perception of the environment and the system itself, and its goals"* [40]. Controlling means mastering the environment of the system as well as the way it is perceived by the SO mechanism, controlling the complete system managed by the SO mechanism, and controlling the goals of the SO mechanism. Observing means to make the system's behavior, its response to the SO mechanism, and the structure of the system accessible and visible. The SO mechanism is highly interwoven with the system it controls, making a testable architecture even more desirable, but hard to achieve. Another aspect that has to be taken into account is that SO mechanisms are often non-deterministic, e.g., since they use particle swarm optimization or non-deterministic heuristics. The non-deterministic behavior makes it hard to control it in a way that a distinct input leads to a distinct output, as Freedman [65] defines controllability.

Approach for Testability of an Architecture for Self-Organizing, Adaptive Systems The Corridor Enforcing Infrastructure (CEI), an architectural pattern for Self-Organizing, Adaptive System (SOAS) tackles the challenge of making an SO mechanism testable. Following the concepts of a feedback loop, the CEI encapsulates the SO (as well as the adaptive) behavior and implementation into a control loop. The control loop consists of defined components, states, and results that are the foundation of the SO. Thus, the SOAS can fulfill its requirements in an ever-changing environment by being reconfigured and controlled according to its needs. The only prerequisite is to provide a Corridor of Correct Behavior (CCB) in the form of constraints that specifies the need. However, the CCB is not mandatory for the implementation of the Self-Organization Mechanism under Test (SOuT). It is possible to test SO mechanisms which have been developed and implemented without explicitly stating the CCB.

The encapsulation enables to address the SO directly, the input and output are specified and observable as well as controllable. The CEI not only provides a testable SO mechanism, but it is also possible to define and categorize failures of SO mechanisms. Besides, we will establish a notion of failure for SO, based on the CCB and the implementation

of the CCB within the CEI. Thus, the CEI will also enable us to complete the set of requirements for testing SO mechanisms.

4.1 Related Work

A software architecture describes components and the subsystems that compose the overall system. Different views are applied to describe the parts of the system and their relationship to manifest the principal design decisions made. Buschmann et al. [23] provide a set of architectural patterns collected from practice and science. Selecting patterns and building an architecture is driven by different goals. Foremost, the aim is to provide specific capabilities within an architecture. These are divided by Buschmann et al. [23] into functional and non-functional aspects. Functional properties, concerning capabilities, directly influence the implementation of the functional goals of the system and non-functional properties are related to other aspects like testability. Testability denotes the ability of an architecture to enable its evaluation. Freedman [65] described testability of an architecture by controllability and observability. These two aspects are of concern in this chapter for the CEI, building an architectural pattern for SO mechanisms. There are further patterns for the architectural relations of SO, mostly with emphasis on functional aspects concerning the adaptation. The most prominent ones are the MAPE cycle introduced by Kephart and Chess [87] and the Observer/Controller (O/C) architecture proposed by Schmeck et al. [143]. The main idea is similar for both, as well as for the CEI: The system is within a feedback loop under control by an SO or adaptation mechanism. Therefore, information is gathered and analyzed (*M*onitor and *A*nalyze by MAPE, *O*bserve by O/C), an algorithm computes a new configuration if necessary (*P*lan by MAPE, *C*ontrol by O/C) and distributes it afterward (*E*xcecute by MAPE, *C*ontrol by O/C). We stick to this feedback loop character for the CEI pattern. As shown by Brun et al. [22] and Sabatucci et al. [140] most of the implemented self-adaptive systems follow this feedback-oriented character. The emphasis of the CEI is incorporating the CCB concept into the architecture and providing testability. We can say the CEI is a non-functional-oriented pattern in contrast to O/C and MAPE. Nevertheless, it is indeed possible to provide functional aspects as well. The fact that many applications follow the principle of feedback-orientation is an advantage since the test approaches based on the CEI are thus widely applicable.

Besides the testability, the CEI is also providing the ability to define the notion of a failure for SO mechanism. This definition further enables to reason about the abilities of SO in more detail, concerning the border of the abilities of SO. Some related approaches are also concerned about testing adaptive systems and thus have their definition of a failure of adaptation. Indeed, adaptation is not SO, but quite related: SO is the adaptation of the organizational structure of the system (or parts of the system). The approach by Fredericks et al. [64] for a definition of a failure of an adaptation is relying on human judgment. For this purpose, the results of the tests are evaluated afterward whether or not they are faulty by a human. Indeed, that limits the number of test cases, since the effort of evaluation is quite large and not automated. Further, no concrete definition of a failure is given. Nguyen et al. [115] are not directly defining a failure. However, they set the pre-/postcondition as mandatory fields of the test case. This prerequisite is enabling

the automatic evaluation, but still involves human judgment for the generation. The approach by Nguyen et al. [115] for testing starts with a test suite compiled by a test engineer, including the specified pre- and postconditions. Afterward, the developed genetic algorithm can mutate the whole test suite automatically. This procedure helps to generate more test cases based on a failure definition for one test case. The degree of automation despite lacking a clear failure definition is thus more significant. Hänsel et al. [74] define a failure based on a model-based description of the target system: whenever it differs from the actual state, i.e., the actual state is no instance of the model description, it is a failure. This definition, however, leaves it to the test engineer to incorporate for example the particular self-healing aspects of the SOAS. We clearly define in general a failure for SO mechanisms in this chapter.

Püschel et al. [127] are offering a taxonomy of failures for the MAPE cycle and show the propagation of the faults. However, the failure definition is too loose to apply it for instance in a test oracle directly. It is more an approach, similar to the responsibilities of SO here, that defines what might go wrong in an SO mechanism.

This chapter adds the missing failure definition for SOAS and further enables testability by providing a testable architecture that can be widely used. Thus, a foundation for testing SO mechanisms is laid.

4.2 The Corridor Enforcing Infrastructure: An Architectural Pattern for Self-Organizing, Adaptive Systems

The CEI is an architectural pattern for SOAS, that is focused on encapsulating the SO as well as the adaptive behavior of the system. Therefore, the CEI is using decentralized feedback loops that monitor and control single components or small groups of components in order to ensure that the system's requirements are fulfilled at run time. The pattern thus is based on the components that are monitoring a dedicated part of the system. Each monitor has a particular responsibility for a condition that needs to be satisfied at run time. A detected violation of the condition is leading to a reorganization of the system (or parts of the system) by a dedicated controller. A constraint describes the condition and its responsibility within a specific context, all the constraints of the system form the CCB. Thus, the CEI is the infrastructure, consisting of feedback-loop-oriented pairs of monitors and controllers which enforce the corridor of the CCB. The CEI describes a decentralized architecture of feedback loops, similar to the MAPE cycle [87] or the O/C architecture [143]. The schematic view in Figure 4.1 shows an implementation of the CEI pattern where the monitor and control infrastructure is based on an O/C architecture. The essential parts of the O/C architecture are:

- The system under observation and control (SuOC), i.e., single agents or groups of agents controlled by the SO mechanism(s).
- The observer (O), i.e., the component monitoring the state of the SuOC and providing information to the controller.
- The controller (C), i.e., the SO algorithms controlling the SuOC.

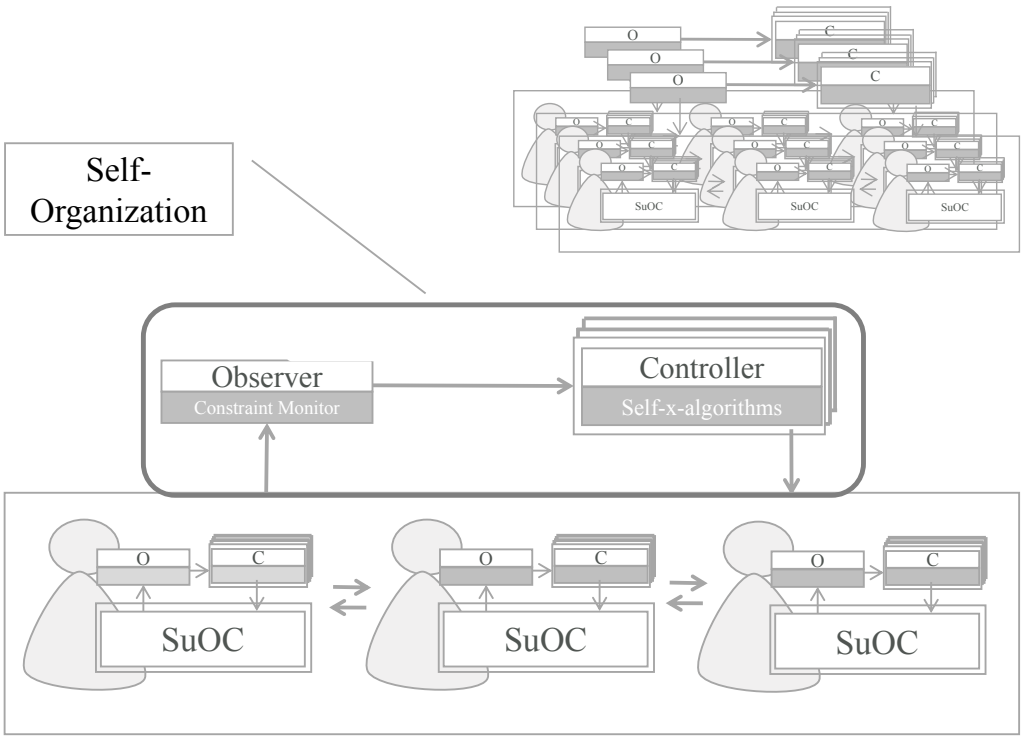


Figure 4.1. Schematic view of the CEI, according to Eberhardinger et al. [52]. The three main components of the CEI are the *Observer (O)*, the *System under Observation and Control (SuOC)*, and the *Controller (C)* that form together distributed, decentralized feedback loops on different levels, i.e., the CEI consists of sets of nested feedback loops (symbolized by the nested set of different system parts in the upper part of the figure) controlling the entire system. If a group of components forms the SuOC, the feedback loop is an SO mechanism.

Note that the CEI consists of sets of nested feedback loops controlling the entire system. The pattern of MAPE as well as O/C can be combined with the CEI pattern, as shown. We differentiate within the CEI pattern between the controlled components by the feedback loops: if a set of components (resp. agents) forms the SuOC, we call it SO (as exemplified in Figure 4.1), if one component forms the SuOC and internals (setting, parameter, etc.) are changed this is called adaptation. For the SO, we call the entire feedback loop SO mechanism, the algorithmic part in the controller SO algorithm, and the constraint monitoring part observer and vise versa for adaptation mechanisms.

Application Case of the Corridor Enforcing Infrastructure Let us consider a realization of the CEI pattern within the smart-grid scenario. The small groups controlled in the smart-grid scenario are the power plants partitioned into Autonomous Virtual Power Plants (AVPPs). Furthermore, single power plants are in control loops to adjust, in particular, their energy production. The reorganization by the controller is performed by one or more SO algorithms resulting in a new system configuration. Such a system configuration has to satisfy the constraints describing valid organizational structures, e.g., a maximum number of controlled components within each organization. Concern-

ing the partitioning problem—applied for the AVPPs—this means that each power plant belongs to precisely one AVPP. For other application scenarios, one might require a structure in which each component belongs to at least one organization. The particular choice of the SO algorithms and their constraints has no impact on the approach, both set covering and partitioning SO algorithms can be implemented within the concept of the CEI.

Testability of the Corridor Enforcing Infrastructure Inside the CCB, the system behaves like a traditional software system and traditional test techniques can be used to ensure the quality of the SuOC. The CEI instead is responsible for the SO and adaptive behavior of SOAS. The testability for the SOAS within the CEI is achieved by the decomposition and encapsulation of the SO as well as the adaptation mechanism. There is a defined responsibility of each observer/controller pair and a precise allocation of the responsibility within each observer/controller pair. The responsibility is defined by a constraint and its context, where the observer is sensing within the context, and the controller is activated via a defined interface and calculates a solution that is distributed to the components within the context. This modular structure and concrete allocation of responsibilities makes it able to decouple the parts of the SO resp. adaptation, equip it with a test harness to make it controllable as well as observable since there are defined in- and outputs. Indeed, the in- and outputs are not trivial, but clearly defined by the context and responsibility.

4.3 Failure Definition and Categorization of Self-Organization Mechanisms

The CEI enables to address the SO mechanism directly via test, since it breaks it down to the major components of SO. These components are responsible for implementing the concepts of the CCB, which describes the obligations of SO mechanism: to restore the INV_{RIA} if it is broken. This obligation is different from a classical invariant—, where a failure occurs if the specified invariant is broken. Consequently, for an SO mechanism, a weaker notion of correctness is needed. The CCB enables to define this notion. Giving this notion is possible based on a more detailed investigation of the phases and responsibilities of an SO mechanism as implemented in the CEI, concerning the meaning of the INV_{RIA}. However, SO is not almighty, not every violation of INV_{RIA} can be restored. The environmental faults can describe the boundaries of SO, that lead to a violation of INV_{RIA} which need to be restored. This is also respected in the failure definition for SO mechanisms, which is formulated by the weaker correctness notion.

4.3.1 Weaker Notion of Correctness for Self-Organization Mechanisms: Definition of Failure

A failure is a deviation of the actual behavior from the expected resp. the defined behavior of software. Our definition of the expected behavior of an SO mechanism is formalized by the CCB. We use that definition as a test obligation. A failure, however, does not occur if the CCB is violated and the INV_{RIA} is broken. The SO mechanism is responsible

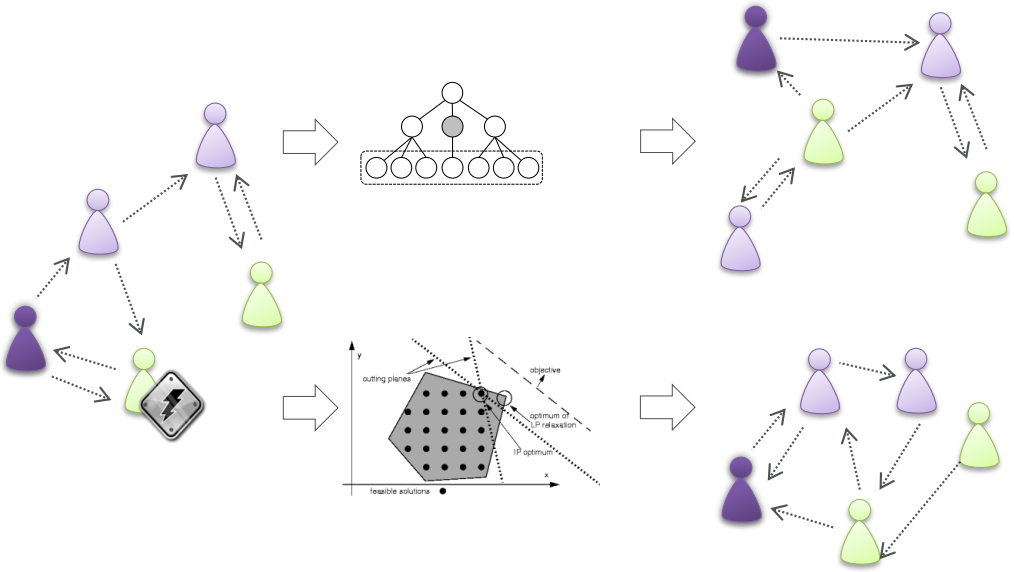


Figure 4.2. The three phases of an SO mechanism are shown: (1) the detection of a necessity to intervene in order to enable the controlled system to continuously fulfill its goals despite the changing conditions, (2) the calculation of a new system configuration by an SO algorithm (here, different algorithms might be suitable as well as different solutions), and (3) the distribution and implementation of the computed solution in the system.

for coping with the situation of an invariant violation, which would be a failure for a non-SOAS, by reconfiguring the system. The completion of this responsibility is illustrated in Figure 4.2 where the three major phases of an SO mechanism are shown:

1. *Detection* of a necessity to take action. This intervention is needed to cope with a situation which would for non-SOAS cause a failure. In the context of the CCB: it is necessary to detect whether the system leaves the CCB. This phase is realized in the CEI in the monitor component.
2. *Computation* of a new system configuration by an SO algorithm, the controller part of the CEI. This configuration needs to fulfill the specification for a valid configuration; the configuration needs to be inside the CCB.
3. *Distribution* of the computed solution, also by the controller of the CEI, to the affected components (resp. agents) of the system which is under control by the SO mechanism.

If these three responsibilities are fulfilled, the SO mechanism works correctly. Thus, the notion of correctness for SOAS is the following, according to Güdemann et al. [70]:

$$\Box \mathrm{INV}_{\mathrm{RIA}} \vee \Box(\neg \mathrm{INV}_{\mathrm{RIA}} \rightarrow \circ \mathrm{INV}_{\mathrm{RIA}}) \tag{4.1}$$

A failure, i.e., Equation (4.1) does not hold, can further be categorized by the responsibilities of an SO mechanisms, leading to this categorization:

1. *Detect* The detection of a violation of the CCB has failed.
2. *Compute* The computed solution is not inside the CCB or no solution is delivered, though there is a reachable one.
3. *Distribute* The computed solution is not correctly distributed, i.e., the system configuration after distribution does not match with the configuration calculated by the SO algorithm.

Testing an SO mechanism is consequently aiming at revealing these failures.

4.3.2 Boundaries of Self-Organization: Tolerable and Intolerable Environmental Faults

Following Equation (4.1), we have included the implicit assumption, that an SO algorithm is always able to find a configuration where INV_{RIA} holds, after a violation has been detected. Indeed, that is practically impossible as SO has boundaries. Thus, Equation (4.1), the failure definition, needs at least a constraint, that states:

if there exists a configuration in the configuration space where INV_{RIA} holds that is reachable from the current, faulty one.

However, it is not straightforward to answer the question of whether or not there exists such a configuration, which might be a valid solution. This question needs to be answered within the automated oracle, forming the non-generic part of the oracle. That part needs to be added to the implementation that is transformed from the requirements model, as described in Chapter 3. Within the realization of the test methods in Chapter 6 we will show how that is done for the described case studies.

An alternative solution might be that a failure occurs if Equation (4.1) does not hold and the system is in a productive state. The productive state is depicted by completion of SO mechanism. If the SO mechanism is not able to find a new valid configuration for the system, it is in a non-productive state. Güdemann et al. [70] called this non-productive state the quiescent state. However, the drawback of this solution is, that setting the system to a non-productive state after each detection of a violation would lead to a correct, but not to a productive system.

In order to setup the requirements as stated above, it is necessary to distinguish situations where SO is powerless and where it has the power to restore INV_{RIA}. We therefore separate *tolerable* environmental faults and *intolerable* environmental faults. Environmental faults are the situations to be considered; situations that are characterized that they change the environment of the SOAS so that it is not able to proceed. These environmental faults are either *tolerable* (such as broken tools in the production cell scenario) or *intolerable* (such as failures of workpiece sensors of the production cell scenario). Tolerable faults are those faults the system can compensate, continuing correct and safe operation after a reconfiguration by an SO mechanism. SO can, therefore, be seen as a mechanism that increases the system's *fault tolerance* in order to prevent hazards for as long as possible. However, SO cannot cope with all faults that occur during the lifetime of a system: Intolerable faults are outside of its reach, either because their occurrences cannot be detected or there is no possible way to react to their occurrence.

In particular, a fault discovery mechanism might be missing due to a deliberate design decision in order to reduce costs or because the discovery is physically impossible for some reason. Additionally, at some point, there is not enough redundancy left in the system to continue operating safely after the occurrence of a tolerable fault, in which case a *reconfiguration failure* occurs. Phrased differently, occurrences of sufficiently many tolerable faults represent the occurrence of an intolerable fault, resulting in a safety hazard. In the case study, reconfiguration fails, for instance, when all tools of the same kind no longer work.

This distinction between tolerable and intolerable faults is necessary for the failure definition since only tolerable faults can be part of the $\mathrm{INV}_{\mathrm{RIA}}$.

4.4 Prerequisites and Benefits for Testing Based on the Corridor Enforcing Infrastructure

So far, the CCB and the CEI are two powerful concepts for making SO mechanisms amenable for testing. The CCB is providing the test obligations for an SO mechanism, enabling to give a failure definition, and derive an oracle. According to Goodenough and Gerhard [177] as well as Weyuker and Ostrand [170], the fundamental prerequisite for testing a system is fulfilled based on this. Further, the CCB provides a clear understanding of the mode of operation of SO. That understanding enables to tailor a testing approach, that is able to reveal failures efficiently by either black-box or white-box testing. Black-box testing is also known as specification testing, i.e., defining the test selection criteria for tests based on a specification; here, provided by the CCB. White-box testing is based on the actual code, i.e., the test selection criteria are defined on code fragments or component elements; here, the needed insights are provided by the CEI. Thus, the CCB and the CEI are a fundament for testing SO mechanisms. Further, the CEI provides an architectural pattern that is testable in the sense of observability and controllability. This lays the ground for building a test harness that can automate and execute the tests and their evaluation. The CEI enables as a result of this to directly associate test cases to SO mechanisms, having the components and their interfaces defined, and capture the results, by having a defined output. Indeed, the input and output are not classical values, like objects or primitives, but the controlled system and its controlled parameter, setting, and configurations. The controlled system acts quasi as input and output.

However, the gains of the CCB and the CEI also sets prerequisites to a testing approach as well as the System under Test (SuT) itself. If the prerequisites were too restrictive, it would imply an approach that is limited to a few systems and thus would limit its contribution.

Prerequisites for the Test Approaches Based on the Corridor Enforcing Infrastructure The prerequisites for a test approach are set by the definition of the CCB: The test obligations are defined as constraints describing valid system states to be maintained by the SO mechanism, as shown in Chapter 3. As a consequence, the definition of a failure, as described in Section 4.3, is used for testing and especially for determining the test oracle. As testing aims in general at revealing failures, the definition of failure further

influences the testing process. However, the application of the CCB should not be seen as a restriction more as an enabler for testing SO mechanisms. Thus, no general test approach is excluded by the prerequisites.

Prerequisites for the System under Test Tested by Approaches Based on the Corridor Enforcing Infrastructure Assuming the testable architecture provided by the CEI goes along with assumptions about the SuT, we assume an SO mechanism with a specific goal to be implemented by the CEI: the SO mechanism is using the information about the system, itself, and the system's environment in order to provide a system's organization in order to enable the controlled system to be productive despite an ever-changing environment. This procedure is implemented in the CEI by a feedback loop with three phases for the SO mechanism: sensing for detecting a need for intervention, computing a solution, and distributing the solution. These phases are assumed to be separated responsibilities that are tested separately afterward. Further, by the constraints of the CCB the monitoring part of the feedback loop is concerning specific properties that need to be accessible. This access also allows for controlling and observing the SuT.

4.4.1 Gain for Testing Based on the Corridor Enforcing Infrastructure

The assumption of having the CEI implemented in the SuT delivers some implications for testing, besides the general controllability and observability. Since the CEI guarantees that all system states fulfill the system requirements, an error-free CEI makes it unnecessary to test concrete system states. Consequently, only the CEI and its mechanisms must be tested instead of the whole SOAS en bloc, which significantly reduces the test effort. Inside the CCB, the system behaves like a traditional software system and traditional test techniques can be used to ensure the quality of the SuOC. The CEI instead is responsible for the SO behavior of SOAS. Hence, we need techniques to examine the CEI and its mechanisms. Fortunately, this can be done separately from the SuOC which consequently reduces the test effort. Test techniques are needed for the responsibilities of the CEI that are defined by the three phases of an SO mechanism: detection, computation, and distribution. Testing these phases can be done separately from the rest of the system which consequently reduces the test effort.

4.4.2 Realizations of the Concepts of the Corridor Enforcing Infrastructure: Application Cases

Overall, the realization of the concepts of the CEI can be broken down to the following:

1. SO is implemented in a feedback loop.
2. The feedback loop has the components for detecting, computing, and distribution; the components are decoupled.
3. The conditions for reconfiguration and successful reconfiguration are specified.

Indeed, there is a limited number of applications that explicitly implement the CEI by design. However, there is a bunch of applications that implement the CEI in principle. Implementing the CEI in principle means that the concepts of the CEI are not explicitly

stated or designed, but still, the system uses SO by a feedback loop and has defined patterns describing a need for SO as well as a successful SO. These principles are named as the generic mechanism for self-adaptation by Brun et al. [22]. The authors describe how adaptation is driven by feedback in general. The authors cluster the feedback into the steps: collect, analyze, decide, and act (which is based on the MAPE loop by Kempart [87]). This clustering corresponds to the CEI, where the detection phase is subsuming monitor and analyze, the computation phase subsumes the decide step, and the distribution phase the act step. There are according to Brun et al. [22] a bunch of systems, especially from the community of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). Two of these systems have been selected as case studies for this thesis (cf. Chapter 2): the self-adaptive web-service system, called *ZNN.com*, and the self-adaptive Apache Hadoop Manager. Both systems are not equipped with an CEI by design, but they have an MAPE architecture, that is mapped directly to the CEI components. Further, the requirements of the systems can be easily translated into an CCB. The concrete steps taken for testing these systems are described within the model-based testing approach of Chapter 6. Sabatucci et al. [140] provide further evidence that the implicit CEI is quite widespread in the implementation of adaptive systems, here SO systems are often subsumed as adaptive systems. The authors developed a general meta-model for adaptive systems resp. SOAS. The meta-model is clustered into four types where the types are extended by increasing type classification, i.e., the type II includes the components of type I and type III the one of type II and type IV the one of type III. The components of type I) are the adaptive system, an effector, an environment monitor, and the world representation. The components are described to work similar as intended by the CEI, the adaptive system and its environment are monitored in order to detect a need for reorganization, the adaptive system computes a new configuration and implements it (distribute if the system is to be distributed). Due to the generalization relation between the types, all types described in the meta-model follow this basic implementation and are therefore categorized as CEI by intention. As the meta-model by Sabatucci et al. [140] is based on a literature review, there are a lot of different systems that can be tested based on a realization of the concept of the CEI.

Summary and Outlook. Software testing makes it necessary to distinguish between the expected output and the actual output, that enables to reveal failures. However, this comparison is often not as obvious as it sounds. We have seen, that the correctness of a SOAS differs from the classical notion of correctness: A system's invariant is repaired by SO mechanism as one duty, and thus it may be violated before if it is restored. This insight is necessary for testing and is one of the contributions of this chapter. The insight is a result from the concepts of the CCB and the categorization of SO mechanisms, presented in this chapter. A further condition is a testable architecture, an architecture that allows for evaluation of its correctness. Therefore, the architecture needs to be observable as well as controllable. The CEI achieves both, an architectural pattern, presented in this chapter. Indeed, there is no such thing as free lunch, the CCB, as well as the CEI, are setting prerequisites to the test approach and the SuT. However, we were able to show, that these prerequisites are not too restrictive and are fulfilled by a large class of SOASs.

Summary. Testing software systems on different integration layers is a common practice [17, 108, 121] for systematically reveal failures. For this purpose, the SuT is disassembled into so-called smallest testable units, which are tested in isolation. These units are afterward integrated and assembled for further testing. Different strategies are available to do so. The most effective one is to compose the system step-wise in a given sequence functionally. The challenges and contributions of this chapter are to identify the smallest testable units for SO mechanisms, to isolate them, to reassemble and integrate them, to find an adequate integration sequence, and to supply the test scaffold for implementing these concepts. The content and contributions of this chapter are published in [46, 48, 49, 52].

5

Isolating and Integrating Self-Organization Mechanisms for Testing

5.1 Related Work **57**
5.2 Disassemble and Isolate Self-Organization Mechanisms **59**
5.3 Reassemble Self-Organization Mechanisms **61**
5.4 Test Architecture for Isolated Testing of Self-Organization Mechanisms . **62**

Decomposition of a System under Test (SuT) and addressing different test requirements, techniques, and methods on different stages or levels is the common practice for testing software [108]. Myers et al. [108] have introduced this classification and decomposition into different test levels, that has been implemented later, amongst others, in the V-Model. Here, testing encompasses the three different stages: system, subsystem, and component level. These levels correspond to different levels of the software development. Thus, it is possible to explicitly match test processes and artifacts to development processes and artifacts. In the V-Model this is graphically notated in a V-like shape, where the development processes and the test processes are on the opposite sides of the V, and the top is completely integrated getting more and more decomposed to the bottom. The aim is to solve the problem of software testing by dividing it into different levels—according to the divide-and-conquer principle. For this purpose, the SuT is systematically disassembled and reassembled for the different tests. All these levels are today relying on automated tests in order to cope with the test requirements for large systems. However, it has been shown (cf. Cohn [39]) that the highly decomposed components of a program are tested more effectively and efficiently than more integrated components. This gain is since tests can be more focused on dedicated functions and are executed faster, as the complex environment is mostly stubbed. These findings are described as the test (automation) pyramid by Cohn [39], that is shown in Figure 5.1. The pyramid is small at the top, and wide at the bottom, that should illustrate the number of tests by the area of the sub-pyramids in the different phases, but also the effort for testing is marked at the y-axis of the drawn pyramid. According to Cohn [39] the following aspects that lead to a high effort are increasing with integration: Tests for integrated systems are

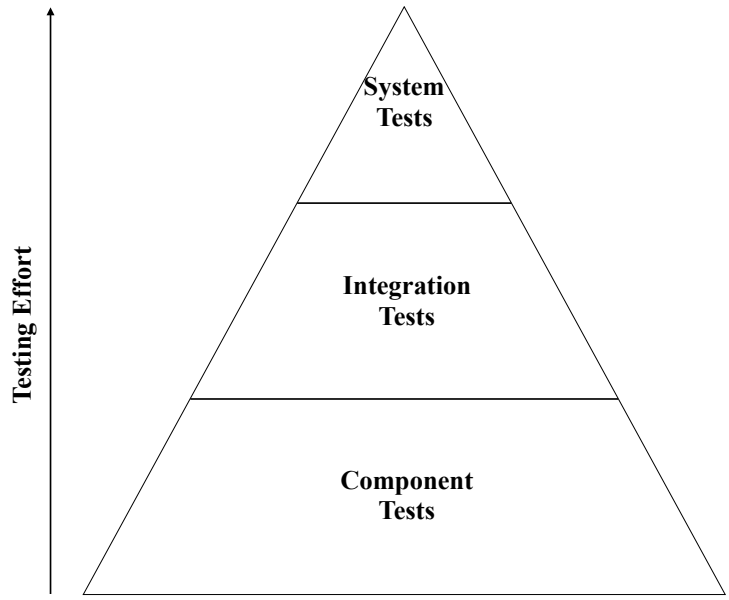


Figure 5.1. The testing pyramid, following Cohn [39], shows the different levels of testing and its corresponding effort. The shown correlation of integration scope and effort is reflected in the number of test cases that are recommended, shown by the area of the sub-pyramids. Thus, the implication is the higher the integration level is, the lower should be the number of test cases.

- Brittle: A small change in the interface of the system can break many tests.
- Expensive to write: Writing tests that can cope with changing interfaces takes time.
- Time-consuming: Tests that run through the system are often taking a long time to run.

Implementing the idea of the test pyramid is demanding good concepts and implementations for the isolation of the SuT. Nevertheless, a system cannot be tested thoroughly only at the lowest level of the testing pyramid, i.e., a Self-Organization (SO) mechanism cannot be test only in the different phases isolated from each other. Some effects, and consequently some failures, only emerge at the integration level. Indeed, this is of particular concern for SO mechanisms, since, the mechanisms are relying on this emergent behavior. Therefore, testing SO mechanisms is divided into three stages:

1. The three defined phases of an SO mechanism (cf. Chapter 4) are tested in separate units,
2. the SO mechanism is tested integrated, and
3. the SO mechanism is tested within the system to be controlled.

An effective integration, as well as an effective disassembling and isolation of testable units, is built on a testable architecture, we rely on the Corridor Enforcing Infrastructure (CEI) (cf. Chapter 4). Disassembling an SO mechanism from the system is based on a clear component-based description of an SO mechanism in the CEI. We disassemble the monitor, the SO algorithm, and the distribution mechanism from the Self-Organization

Mechanism under Test (SOuT). The SOuT is disassembled from the system it controls, that is possible due to the defined interfaces in the CEI. Note, that in most cases more than one SO mechanism is tightly integrated into the controlled system and needs to be extracted. Scaffolding is one of the crucial parts in order to make the SO mechanism testable.

For the later integration different strategies are available for object-oriented software, according to Pezzè and Young [121]: big bang, structural, or sandwich resp. backbone integration. The big bang integration strategy more or less is a shortcut to the system-level, since, the whole system is integrated at once. This strategy is also known as the desperate tester strategy, saying that it is less a rational strategy than a recovery from a lack of planning. Structural approaches are different. Here a plan issues an order for modules that need to be constructed, assembled, and tested in a given sequence. Structural approaches are either focused on features or (as described) on modules that drive the integration. The integration can be either bottom-up or top-down according to the use/include relation of the modules. The top-down strategy starts the integration at the top of the use-hierarchy. Thus, the need for test drivers is reduced. Conversely, bottom-up starts the other way round, reducing the need for stubs or mocks. The sandwich resp. backbone strategy is starting from both ends of the hierarchy.

For SO mechanisms, we follow the feature-oriented approach (which is due to the characteristics of the CEI similar to the module orientation): the different phases of the SO mechanism are combined, and afterward the possible different SO mechanisms are integrated into the system.

In order to be able to test the SO mechanism in three stages, the SO mechanism has to be disassembled and isolated and later integrated stepwise. We consequently have to

1. disassemble and isolate the SOuT into the smallest testable unit,
2. integrate the testable units of the SOuT, and
3. provide an adequate test scaffold to operationalize all that.

5.1 Related Work

Isolation and integration of an SuT for systematic testing on different levels is traced back to the first edition of Myers et al.'s [108] textbook in 1979. Testing an SuT in different levels of integration is an implication of Weyuker's general test axioms [169] and is consequently well-investigated in functional and object-orient programming, cf. [17, 39, 108, 121]. For this purpose, the SuT is disassembled into the smallest testable units and integrated systematically. The modules of an SuT are often selected as the smallest testable units, which are to be tested thoroughly with different proposed test methods to reveal failures from these modules. For objected-oriented software Binder [17], Pezzè and Young [121], and others define the units also by classes. However, even entire module tests are not enough to reveal the potential failures from the SuT, shown by Myers et al. [108] and Weyuker [169]. The consequence is the addition of higher-order tests to the test suite. Higher-order tests subsume integration tests, system tests, and acceptance tests. The test suites and methods on the different levels correspond to the

development level of the SuT, forming the V from the well-known V-Model. We follow the approach of systematically isolating and integrating the SuT.

For isolated testing different methods and techniques have been investigated in the area of isolated testing of component-based systems. An approach in this area, made by Thillen et al. [158], promotes the "tester in the middle" idea that aims at improving testing of distributed components depending on other components in the system. Their application area is the testing of network components. For this purpose, they model dependencies within the network to be able to build mock-ups out of this models. We are going one step further and execute the model for mocking the environment of the tested component, that is possible due to the run time model concepts introduced in Chapter 6. Bauer et al. [14] propose a statistical strategy for isolated testing of component-based systems. Their approach is based on state-based models that are used to generate interaction test models. The goal is to test the interactions and functionalities within the SuT which is composed of several system components. The composition of the components is not straightforward. We are offering an approach in this chapter for addressing this problem for SO mechanisms. Toroi [160] makes a more general approach for improving testing in component-based systems. His work aims to enhance testing methods from the integrator view. Yao and Wang [176] present a framework for testing distributed software components that provides an environment to allow a client-side software component to define tests for a black-box component published on the server-side. The framework focuses on automatic test execution without considering the generation and evaluation of the tests explicitly. Wu et al. [175] propose JATA, a testing language for distributed components enabling the usage of JUNIT in the context of service-oriented systems and offering support for a message-oriented middleware; like Yao and Wang [176] it focuses on the execution of component tests on web services. In contrast to the approaches in [175, 176], the here presented approach is handling another system class with different properties: SO mechanisms. One of the main differences in the properties is the interleaving of the SO mechanism with the controlled system, as described by Tomforde et al. [159], making new concepts necessary.

Integration is the process of reassembling the isolated components of the system during testing. Different levels of integration are specified in the literature [17, 108, 121], where in most cases we differentiated between a fully disassembled system (in modules resp. units), a partially assembled system, and a fully integrated system. The challenge is to provide a test scaffold which can execute test suites on the different integration levels and to find an integration order for the partial integration. The integration order is defining a sequence of components in which they are systematically integrated. For the integration of object-oriented software, there has been intensive research to solve the test integration order problem. The standard strategies, according to Binder [17] are mainly defined over the dependencies of the structure from the classes, e.g., defined by inheritance, associations, implementations, etc. This structure, forming a tree, is integrated bottom-up, starting from the lowest dependencies, top-down, beginning with the control objects, or based on collaboration, starting with the class mostly coupled. All strategies that are introduced by Binder [17] are relying on human judgment. However, there are also automated approaches to solving that problem. Abdurazik and Offutt [2]

compute the sequence for integration. The sequence is built based on the coupling-between-objects-metric—a common software metric for object-oriented software. For this purpose, a graph of all objects is generated, where the edges are weighted with the coupling metric's values. This graph is used to compute the sequences of integration. Belli et al. [16] base their integration strategy on the communication sequence graphs. Within that graph the communication mutants[1] of the system are integrated in order to optimize the selection of the integration sequence. Thus, a heuristic is used to optimize the integration based on covering mutants in the communication by the integration scope. The idea is that different mutants are not able to be revealed if not a particular integration sequence is used. Briand et al. [20] are using evolutionary algorithms for an optimal choice of the integration sequence, based on different possible optimization functions. Winter's [172] approach is also based on solving an optimization problem. The problem is defined by minimizing the dependencies for an integration sequence. We build upon that idea and define an optimization problem for the integration of SO mechanisms, that, for instance, can be solved by evolutionary algorithms or constraint solvers.

5.2 Disassemble and Isolate Self-Organization Mechanisms

Testing on different system-levels of an SO mechanism demands for decomposition. The decomposition is done by disassembling and isolating the SOuT into the smallest testable units. A smallest testable unit is described by the responsibilities, from a specification (resp. black-box) testing perspective, or a defined behavior that can be executed separately, from a functional (resp. white-box) testing perspective. For functional software, the disassembling and isolating is straightforward, as the functions provide a clear point for separation. Indeed, there might also be interdependencies between functions that needed to be stubbed, but the conceptual decomposition is rather simple. For object-oriented software, the disassembling is more sophisticated; Binder [17] dedicated himself to the challenges of testing object-oriented software and comes up with different solutions to the responsibility (he calls it result-oriented) as well as the behavior-based (defined by classes) integration strategies. We extended the strategies for SO mechanisms, as SO mechanisms extend the concept of software that is entirely determined at design time. The challenges arise from the characteristics of Self-Organizing, Adaptive Systems (SOASs):

1. Non-Determinism in the execution of the SO mechanisms, due to the used SO algorithms (e.g., particle swarm optimizer).
2. Ever-changing environment, that is unpredictable due to its complexity and determines the behavior of the SO mechanism.
3. Intense interaction between the system components to be controlled.
4. Concurrent execution within the controlled system and, possibly, concurrent SO mechanisms.

[1] Changing the code of the SuT and rating the test suite how well it can reveal the changes, which are called mutants, is a standard techniques for evaluating the quality of test suites.

For the disassembling follows, that these interdependencies need to be burst. The interdependencies are described by a chain of influences, starting with the ever-changing environment of the SOAS. The environment of the system is the inherent cause for a need for reconfiguration by the SO mechanism. A change of the system's environment causes a violation of a constraint of the Corridor of Correct Behavior (CCB). Indeed, that depends on the definition of the environment, but as we defined it, the environment is consisting of all parts that are not under full control by the system but use or influence the system's outcome. An example from the production cell is the tool of a robot, it is not under full control of the SuT, e.g., the system cannot control whether or not the tool breaks, but it is necessary for the outcome—a processed workpiece by applying tools. Another example from the energy grid: the weather conditions are not controlled by the system, but influence its outcome to produce energy and influence the reliability of the prediction of a power plant. The system, i.e., the system controlled by the SO mechanism, is thus influenced by the environment, whereas the resulting state of the system, as well as the environment, influence the SO mechanism. First, the detection phase of the SO mechanism is influenced by a situation that leads or does not lead to a detection and activation of the SO mechanism. Second, the computation is influenced by the system state as well as by the activation of the detection phase before. Last, the distribution phase is influenced by the system the solution is distributed to and the computed configuration. Knowing the interdependencies enables to break them. The different described parts are the smallest testable units of the SO mechanism, defined by the phases of SO, cf. Chapter 4. The influences, as well as the interfaces defined by the CEI (cf. Chapter 4), enable to isolate them by knowing the behavior to be stubbed. Further, having more control over the smaller parts to be tested, the non-determinism can be limited to the computation unit of the SO mechanisms. The rest of the system is thus made deterministic. The concurrent execution is eliminated for the smallest units as is the intense interaction and the interleaved feedback loops that result from different SO mechanisms in one system.

These units in general are:

- The SOuT's detection mechanism, depending on the environment of the SOAS and the configuration of the SOAS.
- The SOuT's computation mechanism, i.e., the SO algorithm, depending on the detection mechanism, the environment of the SOAS and the configuration of the SOAS.
- The SOuT's distribution mechanism, depending on the SO algorithm, the environment of the SOAS and the configuration of the SOAS.

First, that might sound a bit oversimplified: decomposing a complex SO mechanism to only three generic units, which are to be the smallest testable units. Indeed, there might be smaller units within the three general units. However, these units are most likely to be methods that can be tested classically, as described in the classical testing literature. We focus on the individual needs of SO mechanisms. Thus, the testable units have to reflect the behavior (which is here the same with the functionality, as the behavior drives the functional design in the CEI) of SO. The isolation is described insofar

as a concept. For the operationalization of that concept, a test architecture is needed which technically enables the decomposition by providing the scaffolding. Further, the decomposed parts have to be integrated in order to thoroughly test the system, since, the decompositions might cover errors.

5.3 Reassemble Self-Organization Mechanisms

Integrating the SO mechanism(s) is essential for revealing failures which do not occur in separation. This situation directly follows from the testing axioms of Weyuker [169]: According to the anti-decomposition axiom, a test suite that can achieve high coverage for the system at component scope does not necessarily achieve the same coverage at system or subsystem scope. Thus, testing at system-scope cannot guarantee that components or subsystems have been covered and vice versa. For instance, achieving a specific coverage for the detection component, the computation component, and the distribution component does not necessarily achieve the same coverage for the whole SO mechanism. Following the anti-composition axiom, a coverage at SO mechanism level is not necessarily achieved by rerunning the test suites of the units of the SO mechanism. That is, adequate testing of the units is not equivalent to adequate testing of the SO mechanism and consequently not able to reveal the same failures.

For integration, we follow an incremental strategy, shown to be the most efficient techniques, according to Binder [17]. The argumentation of the effectiveness is based on the systematic of exercising the components in interaction by adding one after another. Thus, the foundation is already adequately tested and the further integration builds upon proven stable interfaces while the integration continues.

Although the incremental integration is conceptually straightforward, the identification of the sequence in which the components are integrated is complicated to identify. This identification is made by carrying out an analysis of the component dependencies. Indeed, the dependencies and the whole integration process is linked to the architecture of the software. We rely here on the CEI. The decisive dependencies are here the influences and interfaces between the units for detection, computation, and distribution, as described above. These phases form the first integration step. The second one is formed by the collection of the SO and adaptation mechanisms in the system if more than one is available. Thus, we have different building blocks for the composition, and at every integration step, one block is added. Every block is described by its dependencies. The set of dependencies of a block is every interaction outside the block, often called fan-in [172]. For instance, an SO algorithm is interacting with the monitoring unit, the controlled system, and the distribution unit, forming the direct dependencies. The interaction with the environment and other SO or adaptation mechanisms is indirect via the controlled system. For isolated testing a stub is needed for every direct dependency, the direct dependencies can thus be diminished by integrating other blocks, e.g., integrating the SO algorithm with the distribution unit. Indeed, the different dependencies have a different effort for the test scaffolding, and maybe an integration may lead to a larger dependency in the newly formed block than in a single block before, but the union of both is at least the same size, however the aim should be to have a smaller one. This

information is used to form a cost function, according to Winter [172]. Therefore, we use the following optimization problem for integration, adapted from Winter [172]:

$$\begin{aligned} &\underset{o}{\text{minimize}} && C(o) = \Sigma_{i,j=1}^{n} cd(i,j) + cid(i,j) \\ &\text{subject to} && c(i) < c(j). \end{aligned} \tag{5.1}$$

Where $o \in \sigma$ is the computed integration order having σ as the set of all possible permutations of the integration, and $c(i) = k$ with $i, k \in 1, \ldots, n$ the assignment of the integration position of the ith numbered component (assuming we have a complete numbered order of all smallest testable components), $cd(i,j)$ the cost function for the direct dependencies between i and j, and $cd(i,j)$ for the indirect dependencies between i an j. The cost functions have to be supplied by a test engineer, and the dependency graph needs to be cycle free. Having Equation (5.1), a general purpose constraint solver like *IBM ILOG CPLEX* [1] can be used to solve the integration problem, as shown by Winter [172]. The result is a sequence optimal to the indirect and direct dependencies. That sequence is optimized for the test stubs and mock, which turned out to be most challenging in the technical implementation for the test scaffolds of the case studies discussed in Chapter 2.

5.4 Test Architecture for Isolated Testing of Self-Organization Mechanisms

So far, we have conceptually separated and integrated an SO mechanism for staged testing. The isolation and disassembling are based on the CEI pattern. We identified the units of an SO mechanism which are tested in separation and afterward systematically integrated into the SOAS: the detection, computation, and distribution unit. In order to enable testing as described a test scaffold is needed. A test scaffold is an additional code needed for the execution of a test suite on isolated, partially integrated, and completely integrated SuTs. The three main components of a test scaffold are the test driver, the test stubs, and the test oracle.[2] The test driver is responsible for executing test cases from the test suite, the test stubs are in charge of simulating the environment, i.e., all components the SuT is isolated from, and the test oracle is responsible for the evaluation of the test cases, as described in Chapter 3.

The effort for providing the test scaffold is often underestimated. According to Pezzè and Young [121], the average effort spent on providing the test scaffold is almost half as much as for the complete SuT. Consequently, it is worth to engineer the test scaffold systematically. For this purpose, a test architecture is designed and implemented in this thesis. The test architecture is shown and described in Figures 5.2 and 5.3. Figure 5.2 shows the structure of the test architecture, it is separated into a `Test Suite Generator`

[2] Often in literature, cf. Pezzè and Young [121], a test harness is mentioned as an additional component of the test scaffold. The test harness is used for providing the deployment environment of the SuT. As for SO mechanisms the functional environment of the SuT and the environment where the SO is deployed are amalgamated, we use only the term test stub. Further, we have a completely automated test oracle, as described in Chapter 3, there is no use for mocks, which are enhanced stubs which are also responsible for test case evaluation.

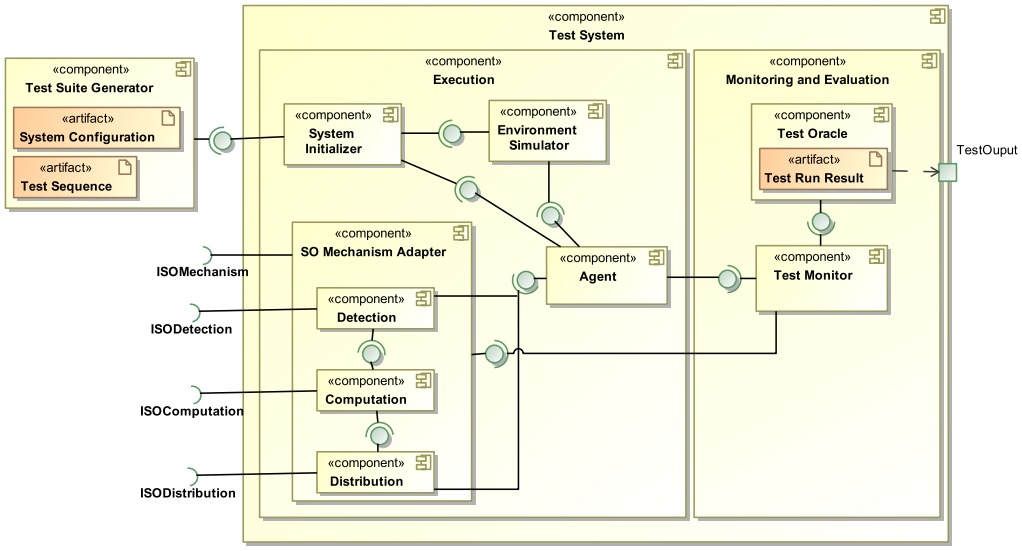


Figure 5.2. The shown Unified Modeling Language (UML) Component Diagram describes the general architecture for the test scaffold used for testing SO mechanisms. It is separated into the three main components for full automatization: the `Test Suite Generator`, the `Execution`, and the `Monitoring and Evaluation` components. For testing, different parts of the SO mechanism or the whole SO mechanism is plugged into the system via an interface. The adapter enables to execute the test suite by simulating the environment and the controlled part of the SO mechanism as `Agents`. Further, the adapter enables to monitor the SuT and use that information to judge over the results of the test execution by the `Test Oracle`.

and a Test System. The Test Suite Generator is responsible for supplying System Configurations and matching Test Sequences to be executed, forming together the test suite for the SuT. Indeed, the level of isolation resp. integration is influencing the needed information provided in the test suite. This information concerns the integrated components and the not integrated components, and is supplied for the Test System by the System Configuration. It shows which part of the SuT is accessed via test drivers and which part is mocked by test stubs. Both, test drivers and test stubs, are integrated into the Agents and the SO Mechanism Adapter. The Test Sequences are describing environment changes, simulated by the Environment Simulator. They are independent of the degree of integration. The Execution component, within the Test System, is responsible for setting up a test suite and executing it. For execution, the Environment Simulator is manipulating the state of the Agents in the Test System according to the test cases described in the Test Sequence. The Agents are the controlled system of the SO mechanism. They are initialized and set up by the System Initializer. For this purpose, complex properties of the real system are condensed to more straightforward value representations. The mechanism of these complex properties is incorporated into the test case generation, as shown in the next chapters (Chapters 6 and 7). This parting supports a clear separation of concerns: The test stubs and test drivers are responsible for feeding the input into the SuT and have to be able to make different test levels (integration levels) accessible. The test generation needs to be able to describe the complex process of creating different inputs. For instance, a solar power plant might be represented by an Agent that only has a numeric value for its state (depicting the credibility of energy production), if that is the observed variable of the SO. This abstraction is done by the test engineer within the test case generation (either manually or, like proposed in Chapters 6 and 7 in a model-based fashion) and described in the System Configuration by different types of Agent that need to be created as well the properties. The change of a property is then described in the corresponding Test Sequence.

The SuT is plugged into the system via an interface. Depending on the level of integration of the test to be executed different components are integrated and tested. The SO Mechanism Adapter stubs the not integrated parts of the SO mechanism. Therefore, the stubbing behavior needs to be provided within the SO Mechanism Adapter. Thus, the external behavior in the Execution component of the SO Mechanism Adapter is always the one of a complete SO mechanism. The Monitor and Evaluation component observes the test execution via different interfaces by the Test Monitor and evaluates the data by the Test Oracle, that is implemented as described in Chapter 3.

Figure 5.3 shows an UML Sequence Diagram of the test architecture, which describes the dynamic behavior of the architecture in more detail. We use this architecture for the further implementation of the test systems in this thesis and will illustrate how it is implemented for different test approaches and different SuTs. The architecture reflects the nature of SO mechanisms by emphasizing the environment and the controlled components of the SO mechanism. Simulation is for this purpose an essential part for stubbing the not integrated parts of the SuT. The simulation enables to provide inputs to the SuT, which are described in the Test Sequence. As described in Figure 5.3,

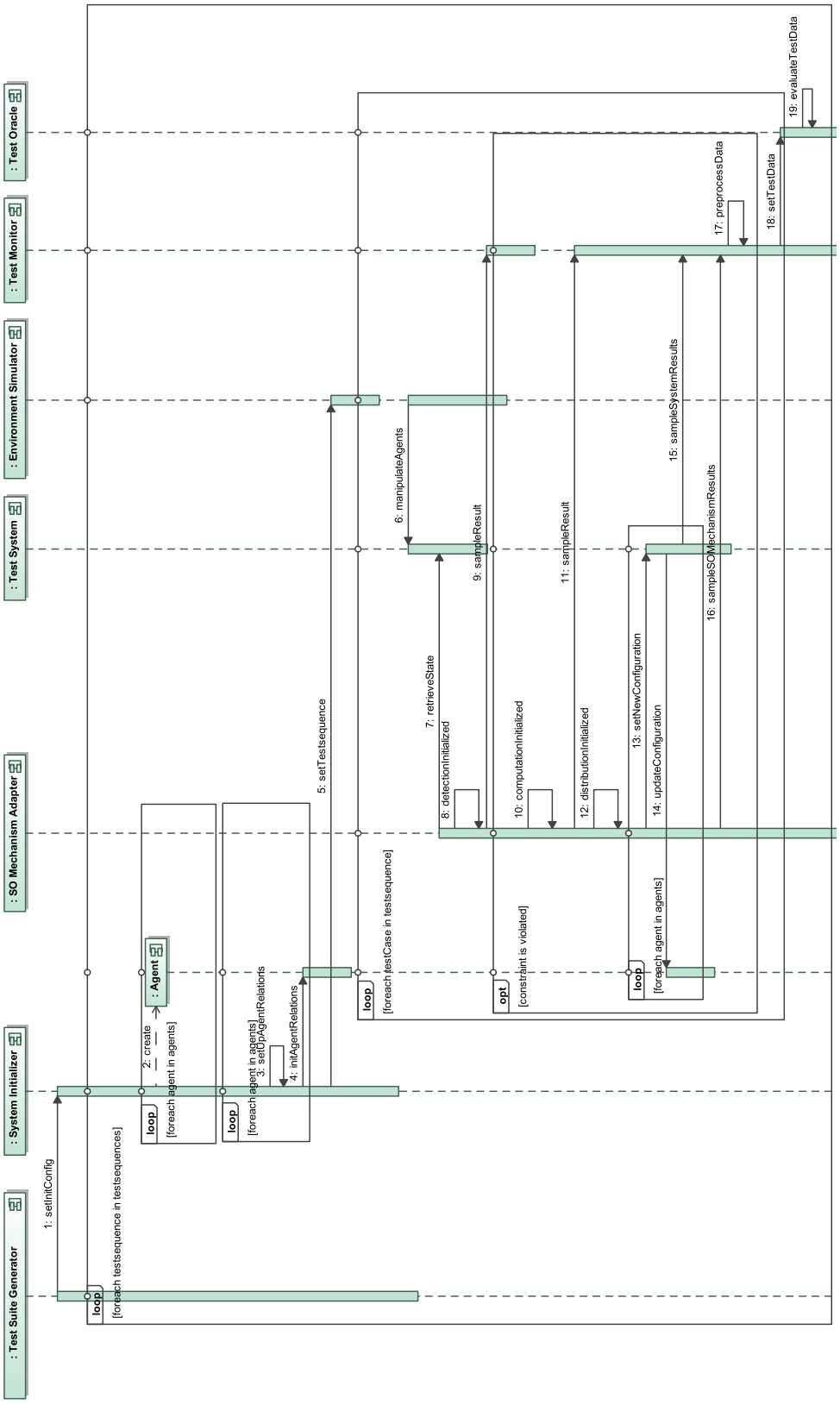


Figure 5.3. The shown UML Sequence Diagram describes the abstract dynamic behavior of the execution of one test suite in the developed test architecture, shown in Figure 5.2. The described steps will take several times for different `Test Suites` generated. First, the system is set up and initialized according to the `System Configuration` of the applied test suite. Afterward, the given `Test Sequence` is executed by simulating the environment and manipulating the `Agents` within the system. The `SO Mechanism Adapter` is running through the different phases and either simulates a part that is not supplied or passes the information via the interface to the SuT. The `Monitor` is sampling data at the neuralgic points of the SO mechanism (at different points of the interaction). This information is evaluated by the `Test Oracle` to form the `TestOuput`.

each Agent needs an initialization according to the configuration of the test suite. Afterward, the test cases, within the Test Sequence are executed by setting them for the Environment Simulator. The setup involves in general, as shown in Figure 5.3, the Test System's stubs, but also the configuration provides information about the SuT, its composition, and configuration. The SO Mechanism Adapter is the centerpiece of the architecture, enabling to integrate the SOuT, but also to stub not integrated parts. The steps 8, 10, and 12 of the UML Sequence Diagram in Figure 5.3 can be either used as a stub or test driver. These steps correspond with phases of an SO mechanism, as described in Chapter 4, and are the neuralgic points for the Test Monitor to sample data for later evaluation, as shown in the steps 9, 11, and 13 in Figure 5.3. All that information is collected by the Test Monitor and processed with the Test Oracle.

Summary and Outlook. Theory [169] as well as practice [17, 121] have proven staged testing of software systems as the most effective and efficient method for revealing failures. Testing of SO mechanisms is no exception, here the divide and conquer principle and the dedicated examination of the units of the SO in the SOAS is needed to reveal failures. As we will show in Chapter 6 failure masking—an aspect to be considered due to the self-healing capabilities of SOAS—is only identifiable in isolation, whereas some other errors are only propagated to a failure in interaction and thus integration. We set the stage to enable different test techniques to act at different test stages in this chapter. We explained how to disassemble an SO mechanism despite its tight integration in the SOAS and its interleaving with other SO mechanisms. We showed how a disassembled SO mechanism is testable in isolation. The concepts for isolation of SO mechanisms are needed for fully disassembled SO mechanisms as well as for partly integrated SuTs. Integration of SO mechanisms is based on the identified dependencies, which are either direct or indirect. The indirect dependencies are responsible for the emergent behavior, that is characteristic of SOAS. Having the dependencies, identified by a test engineer, a constraint optimization problem can be formulated and solved for computing the integration sequence for an SO mechanism with the minimal dependencies, which need to be stubbed. Stubbing is beside the test driver and the test oracle the primary responsibility of a test framework. In order to enable a systematic development and implementation of a test framework, we supplied an architecture that is used in this thesis for all further testing approaches and is applied to all case studies.

Summary. We establish a closed-loop Model-Based Testing (MBT) concept for SO mechanisms. Therefore, the information from the executed SuT and its environment is fed back into an executable run time model. Thus the model is used for the evaluation of test results, the provision of information for test case generation, and the execution of the tests by executing the models. We differentiate test models for continuous and for discrete SO mechanisms. For continuous SO mechanisms, the focus is on describing the environment based on probabilistic profiles, which are used for test case generation. A fault-based test model, describing environment faults as tests, for discrete SO mechanisms is further provided. We show how Back-to-Back (BtB) testing is used for providing high-quality test models. Afterward, we evaluate the approach in the context of five different case studies. The content and contributions of this chapter are published in [45, 48, 49, 51, 52, 57].

6 Closed-Loop Model-Based Testing for Continuous and Discrete Self-Organization Mechanisms

6.1 Related Work **69**
6.1.1 Run Time and Design Time Approaches for Testing Adaptive Systems 70
6.1.2 Model-Based Testing 71
6.1.3 Back-to-Back Testing 73
6.2 Closing the Loop of Model-Based Testing **73**
6.2.1 Feedback in Model-Based Testing 75
6.2.2 Concept of Run Time Models 77
6.2.3 Model Reflection for Reflecting Changes in the System under Test . . 78
6.3 Probabilistic Models for a Continuous Self-Organization Mechanism . . **79**
6.3.1 System Model 79
6.3.2 Environment and Test Model 81
6.4 Fault-based Testing Models for Discrete Self-Organization Mechanisms **85**
6.4.1 The System Model for Discrete Self-Organization Mechanism 86
6.4.2 The Environment and Test Model for Discrete Self-Organization Mechanisms 86
6.4.3 Designing Test Models with Environment Faults 88
6.5 Back-to-Back Testing of Test Model and Implementation **90**
6.5.1 Using Executable Run Time Models for Back-to-Back Testing 91
6.5.2 The Special Case of Back-to-Back Testing Self-Organization Mechanisms 91
6.6 Evaluation **92**
6.6.1 Production Cell—Testing an Integrated, Discrete Self-Organization Mechanisms in a Back-to-Back Test Setting 94
6.6.2 Energy Grid—Testing a Disassembled, Continuous Self-Organization Mechanism 103
6.6.3 Load-Balancing Web-Service—Evaluating the Test Approach in a Controlled Experiment 117
6.6.4 Pill Production—Investigating Reusability and Generalizability of the Test Model in Resource-Flow Systems 121
6.6.5 Apache Hadoop—Testing an Industrial Case Study in Full Integration . 123

So far, we have developed an approach for specifying the behavior of Self-Organization (SO) mechanisms, automatically derived a test oracle, designed a testable and widely applicable architecture, where we showed how it is used to isolate and integrate the SO mechanisms. In this chapter, we will use all these foundations for actual testing the SO mechanisms against their specification in a Model-Based Testing (MBT) fashion. When we say testing we mean the execution of the System under Test (SuT) under different conditions with the intention of revealing failures. For this purpose, a test suite has to be generated, executed, evaluated, and rated according to its adequacy. We will show how that is possible in full automation within the MBT concept.

The MBT concept enables to handle the complexity of the SO mechanism by abstraction. For this purpose, we will build the overall model from three different parts: the system model, the environment model, and the test model. The system model is used to describe the SO mechanism itself. The environment model describes its dependencies to the environment as well as the environment itself. The test model enriches both models with information for deriving test cases. Further, the model even enables to script the test suite and execute it on the SuT, by using the model as a test scaffold.

The environment is of utmost importance for SO mechanisms, as it is determining its abilities. As the environment of SO mechanisms comes in different shapes, as described in Chapter 2, different modeling approaches are needed for the test and environment model of discrete and for continuous SO mechanisms. Thus, for discrete SO mechanisms, discrete changes of the properties of the environment are to be modeled and for continuous SO mechanisms, continuous property changes are described. For the first, we rely on the concept of fault-based testing applied to the environment of the SuT, annotating possible faults into the controlled environment of the Self-Organization Mechanism under Test (SOuT). A possible environment fault might be the following: a drill in the production cell is broken, i.e., a particular environment fault, and consequently is no longer available to the Self-Organizing, Adaptive System (SOAS) and thus changes the solution space of the responsible SO mechanism. For the continuous SO mechanisms, the model is a probabilistic description of the continuously changing environment, this is called Environment Profile (EP). An EP is a probabilistic description of state changes in the environment. For instance, the changing prediction quality of a solar power plant is described by a continuous variable, that is changing each step[1] by a defined change described in the model with a defined probability. These environment models, both for discrete and continuous SO mechanisms, are used for the test case derivation. This is done in a so-called online testing approach, i.e., the test cases are derived, completed with test data, and executed right after another. Since we use executable models, the test execution is the execution of the model that is producing an output that could be fed into the SuT as a test input. Besides the environment of the SuT, the SuT itself is part of the model. That makes it easy to integrate different parts of the SuT since parts of the model change their roles, i.e., the test stub becomes a test driver for the SuT and vise versa. Further, the generated test oracle, from Chapter 3, can execute its

[1] Note that we described in Chapter 5 a step-wise execution model. Thus, although the properties of continuous SO mechanisms are changing in fixed time steps the value is still continuous.

constraint evaluation on the test model that is instantiated with the current state of the environment and the SuT. For this purpose, the models need to evolve with the system and reflect the current state of the system. We describe how that is possible with the concept of run time models.

Testing the test model and its scaffolding is necessary since the complexity of the test framework, i.e., the complete environment for testing, is vast and human faults may also occur in designing and implementing the test setting. We show how this is done by applying Back-to-Back (BtB) testing [166]. For this approach, the test engineer and the development engineer are developing the actual system and the test framework back-to-back. That enables to reveal inconsistencies by executing the developed system with the test framework. The test engineer and the developer have to decide whether the test system or the SuT is incorrect according to the specification.

The whole approach of MBT for SO mechanisms, as presented in this chapter, is thoroughly evaluated. In Chapter 2, we describe how five different systems have been tested. The evaluation demonstrates different capabilities of the approach presented in this chapter. Two of the systems, the energy grid (with a continuous SO mechanism) and the production cell (with a discrete SO mechanism) are investigated in depth. However, all described case studies are thoroughly tested, we use them to show different aspects of the approach and consequently put different emphasis on the details. Showing a detailed version of the probabilistic model and the fault-based model is the starting point of the evaluation. As proposed in Chapter 5, we start by disassembling the system and test the SO algorithm in isolation, this is shown for two different SO algorithms within the energy grid. Next, we show how to integrate the SO mechanism of the production cell for testing. The full integration is shown for the Hadoop system, here we switched all stubs to test drivers and can execute the test cases on the system-level. Further, the Hadoop testing endeavor allows us to judge over the abilities of the approach to be applied to industrial systems. The web-service case study is used with emphasis on the usability of the approach. As the implementation was carried out in a student project, we discuss our observations from this point of view. Last, the pill production demonstrates how generic the approach is by relying on generalization. The pill production, as well as the production cell, are implemented on the basis of the same meta-model for resource-flow oriented systems.

6.1 Related Work

The necessity of testing adaptive systems has been recognized both in the testing community [114, 118, 173, 174, 180] and in the community of adaptive systems [40, 63, 127]. Siqueira et al. [148] present an overview of the different concepts and techniques for testing adaptive system given in the literature. They summarized non-determinism and the emergent behavior as the main challenges for testing adaptive systems claimed by their investigated papers. Testing SO mechanisms face similar challenges, caused by moving decisions from the design time of the system's development to the run time of the system's execution. In general, the approaches for the quality assurance of adaptive systems could be clustered into run time and design time approaches. The idea is to

use similar concepts and ideas for testing as for designing the SuTs. The design time approaches focus on assuring the quality before the system is released, in most cases via testing. The run time approaches instead are carrying out the quality assurance after the release, that is what in most cases is called at run time in the related approaches. The approach for quality assurance in this chapter is executed at design time but makes use of run time concepts, but denoting run time with the execution time of the test suite. We show how MBT could be used to assure the quality of SO mechanism.

The review of the related work focuses on different areas: we will discuss in general the design and run time approaches for testing adaptive system, as the approaches are entirely related to testing of an SO mechanism. Further, we are reviewing the approaches in the field of MBT, which is the central methodology in this chapter, as well as the work of the model@runtime community. Besides, the approaches of BtB testing are also discussed in this section, relating to the concept of BtB testing of SO mechanisms as proposed in this thesis.

6.1.1 Run Time and Design Time Approaches for Testing Adaptive Systems

All, run time as well as design time approaches, presented here are focused on assuring the quality of adaptive systems. Siqueira et al. [148] summarize the work in this area and show the main challenges, namely non-determinism and emergent behavior which are due to autonomous decision making at run time. The adaptive systems are, in contrast to the SO systems, focused on the adaptation of parameters, settings, and capabilities of the system by (in the most cases) a central instance. Self-organization instead is adapting the organizational structure of the system or parts of the system in (most cases) a decentralized approach. The difference is the integration of the mechanisms and the abilities to change the system. Nevertheless, the approaches for quality assurance are still related to the work in this thesis.

Run Time Approaches Run time approaches for testing take up the paradigm of run time verification [59, 60, 90]. They shift testing into run time to be able to observe and test, e.g., the adaptation to new situations. Camara et al. [26] are using these concepts to consider fully integrated systems. Their testing approach focuses mainly on testing non-functional properties of the system or more precisely, the resilience of the adaptive system. The authors, therefore, investigated the system's adaptive capabilities by collecting and analyzing data in a simulated environment. The gained information is used as feedback for the running system. Ramirez et al. [130] take a similar approach, that is also focused on non-functional requirements. The authors use the sampled data from a simulation to calculate a distance to expected values derived from the goal specification of the system. This information is used to adapt the system or its requirements proactively during run time. Run time approaches, however, are limited to tests of the fully integrated system and therefore are faced with problems like error masking which is very likely in such self-healing systems. In this thesis, we benefit from the piecemeal integration of the system for testing relying on the Corridor Enforcing Infrastructure (CEI). Thus, it is possible to avoid error masking within this approach by testing the SO mechanisms in isolation.

An essential difference to the mentioned work is that the approach of this thesis is using testing for revealing failures instead of analyzing the current system state for generating feedback for the adaptation. Still, we also use the basic concepts of run time testing. The CEI allows us to split the evaluation into the three responsibilities Detect, Solution, and Distribution which in turn enable us to evaluate the runs without the evaluation of complex system states on the system-level. As the evaluation shows, the CEI-based testing approach is especially beneficial in the context of self-organization.

Design Time approaches Design-time approaches like [92, 114, 118, 155, 180] test the systems during its development. All of these approaches are considering some dedicated parts of the system. Consequently, it is not possible to give evidence about the correct functionality of the overall system. Zhang et al. [180] compose their tests towards a fully integrated system test, but they do not consider adaptivity or SO explicitly since they focus on testing the correct execution of plans within multi-agent systems. Nguyen et al. [114] promote an approach for a component test suite (where the components correspond to agents) but do not consider interaction or organization between the agents as it would be necessary for SO.

The evaluation of the test results, i.e., the application of a test oracle for adaptive behavior, is only considered by Fredericks et al. [63, 64] and Nguyen et al. [116]. Both approaches are relying on goals reflecting the requirements of the system that are somewhat loosened in order to reflect the ever-changing environment the agents have to adapt to: The approaches mitigate the goals with the *RELAXed* approach [171] or consider soft goals that do not need to hold at all times. Consequently, the decision of the test oracle is somewhat fuzzy. In the approach proposed in this thesis, the definition of correct and in-correct behavior is given by the Corridor of Correct Behavior (CCB), that enables us to decide whether a failure occurs or not.

6.1.2 Model-Based Testing

Using models, in an implicit or an explicit form, is a widely used testing technique, that has been proposed by different authors [17, 21, 68, 126, 162] for easing and structuring the process of testing. Model-Based Testing is used for reducing the complexity by the abstraction ability of models. Pretschner and Philipps [126] summarize different areas of applications and methods that are all commonly known as MBT. However, none of these cases are using the information of the executed tests to provide feedback within the model; this is what is proposed in this chapter as closing the loop of MBT. Further, the focus is mostly on generating test models by describing abstract test cases as a model, e.g., by using an Unified Modeling Language (UML) sequence diagram and deriving a set of concrete test cases based on a data set of test data. The use of models in this chapter highlights the environment and includes different concepts of test case description. All these concepts and extension serve the purpose of testing SO mechanisms. One crucial aspect is closing the loop and using the model at run time. The aspects of using models at run time are also investigated in the models@runtime community.

Models@Runtime Aßmann et al. [12] describe the concept of models@runtime. The main idea is to instantiate the model at run time and also to make intense use of different types of models for improving the system while it is executed. In general, the concepts are in the most cases applied to enhance the modeled system at run time by using the prediction or computations for new configurations of the system [12]. Trapp and Schneider [161] investigate the usage of models at run time for safety certification with similar approaches like the one introduced for run time verification by Leucker and Schallhart [90]. In this case, the evaluation of the properties is executed on the model, similar as done in this chapter for testing, and if a property fails the safety of the system is no longer given, and some sort of quiescent state is needed, as described by Güdemann et al. [70]. Habermaier [71] as well as Leupolz [91] applied the concepts of models@runtime for the verification of safety properties. They used the abilities of the executable models to check its state space via executing every possible state in the model. This execution is a form of model checking for safety properties. In this context, the modeling language S# has been developed, which is a language used to describe run time models in this thesis.

Test Models of Environmental Changes There are several approaches to tackle uncertainties about the expected usage of an SuT within testing models. They can be summed up under the idea of *operational profiles* [149]. The information within these test models represents the user's behavior in a probabilistic model. For this purpose, different techniques for generating and using these profiles have been provided.

Operational profiles thus are an established technique for modeling uncertain behavior—mainly of the user—for designing test models and for evaluation purposes. In this chapter, we use environment profiles which are based on a similar concept to deal with the complexity of the ever-changing environment of SOAS by reducing its state space using a probabilistic approach.

Sammodi et al. [142], e.g., generate usage profiles, as they call them, by monitoring the user's interaction with the system and deriving the profiles for the observed usage afterward. One of our possibilities to establish EPs also follows this monitoring and analysis process with the difference that we are not monitoring users of the system. Instead, we monitor the whole system environment which includes all influences on the SOuT. Samih et al. [141] present another approach to design models of the usage. They enrich the models by introducing capabilities in order to model variants of specific features, i.e., product features, to form test models for product line engineering. The approach made by Ehlers et al. [58] focuses on using the usage profiles for detecting anomalies in adaptive systems in order to use the information about the anomalies during the adaptation process. Besides focusing on handling user behavior, there is also some work on representing the behavior of other system components in the test model, like Popovic et al. [125]. The authors use the models for protocol testing and therefore represent valid and invalid communication between components.

6.1.3 Back-to-Back Testing

Back-to-Back testing was initially proposed by Vouk [166] and describes the concept of the co-development of a test system and the actual system or mechanisms based on the same requirements, letting the two systems compete with each other in order to reveal discrepancies and errors. Back-to-Back testing emphasizes the correct interpretation of the actual requirements and their implementation. The assumption made is that two different developers resp. development teams will not make the same mistake twice, i.e., misinterpret or neglect functional requirements, and so the discrepancies between the two systems reveal potential development errors. This concept is in this thesis applied to the development and test of SO mechanisms.

6.2 Closing the Loop of Model-Based Testing

Testing aims at revealing failures, i.e., showing that the actual and the intended behavior of the SuT differs. For that purpose, the test engineer has to get clear about the intended behavior of the system. Test engineers use the requirement specification of the SuT to gain the necessary understanding of the intended behavior. We could say the test engineer builds a mental model of the system's behavior, requirements, and of possible test cases. Binder [17] argues that consequently, all testing is model-based. Indeed, this approach is *'implicit, unstructured, not motivated in its detail, and not reproducible"*, according to Pretschner and Philipps [126]. Model-Based Testing makes these implicit models explicit. Thus, MBT is used for structuring the testing approach and its activities and further, allowing for automation of testing activities by using models describing either abstract test cases or the system's requirements and behavior. The focus of the test engineer can be shifted from implementation and execution of tests to the design and analysis. This is possible as the test models are used for [162]:

1. Test data generation
2. Test case generation
3. Test case generation with a corresponding oracle
4. Test script generation

Focusing on design and analysis is of special interest for large and complex systems, that is where the additional effort for the creation of the models pays off [21, 131]. However, models are easier to understand, validate, maintain and are useable for automation making the initial effort paying off [126]. Therefore, the model requires to be more abstract than the SuT and/or concrete test cases and needs a defined syntax and semantics to be machinable for automation. Abstraction is an inherent property of models following their definition: A model is, according to Stachowiak [151], defined by the following properties:

1. The *mapping* property: Models are mappings from the concrete (original) into an abstract representation.

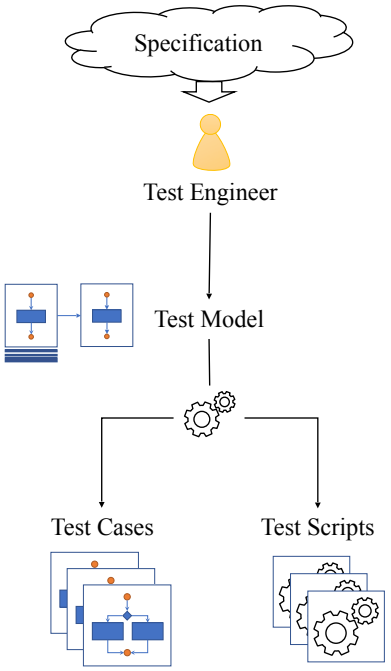


Figure 6.1. The generic MBT process, according to Götz et al. [68]. Starting position is the specification, the test engineer uses it for generating a test model. The test model comes in different shapes, either abstract test cases and test data or a description of the intended behavior of the SuT. The model is used for the automatic generation of test cases and/or test scripts to be executed on the SuT.

2. The *simplification* property: Models are not reflecting all attributes of its representation (original), only those which are relevant for the corresponding usage of the model.
3. The *pragmatism* property: Models serve a specific purpose.

Abstraction is achieved by omission and/or encapsulation of details within the model [126]. The details of the concrete systems are omitted to get rid of clutter, i.e., information that is deemed irrelevant for the specific purpose. However, if missing information is considered as necessary, it should be refined by the test engineer. The omission is needed for providing the intended properties of the model, which supports handling complex systems. Encapsulation of details further supports these properties by reducing the complexity by incorporating references and not the content they stand for [126]. Indeed, omission and encapsulation of details are two related activities. Test models might omit details for getting the intended simplification of the SuT, used for generating test cases. However, to execute the test cases more concrete information might be needed; we need the encapsulated detail to fill in this information. We are consequently working on a different level of abstraction.

Figure 6.1 illustrates the process of MBT from system specification to test cases and test scripts. The responsibility for assembling the test model is due to the test engineer, who needs to find the right abstraction for the model. Once the test model and the toolchain

for automatic derivation of test cases and test scripts is established it is possible to easily add or change aspects within the model, for instance when the specification is changing. The focus is set on the design and specification of the test suite which comes with a bunch of benefits. First of all, the test models enable an in-depth understanding of the specification of the SuT, making it explicit in the model. That model is traceable in the sense that each part of it is connected with the specification, but also with the generated artifacts, like the test cases and the test scripts. The test model, in contrast to an implicit model of a test engineer, can be validated, i.e., checking whether the test model conforms to the specification, but also whether the specification tackles the actual needs of the users. Thus, the test model is well-suited to define and check the adequacy of the desired test suite on it [162]. This integration of different vital aspects helps to improve the quality of the test artifacts, as demonstrated by Utting and Legeard [162]. For this purpose, there is an implicit closed-loop where the model is giving feedback to the test engineer, but also to the specification. That is of importance when the SuT and its specification is complex, as it is for SOAS, to achieve the desired high quality of the system.

However, the process from the test model to the SuT and its environment as well as the execution of the test cases is directed one-way (cf. [68, 126, 163]), as shown in Figure 6.1. For SOAS, this feedback is needed in order to cope with the self-organizing behavior. Having a closed-loop MBT approach enables to use the feedback for enhancing the test model. The model has now the ability to adapt to the SuT and could be used for evaluations and decision making at run time. Figure 6.2 shows the process of Figure 6.1 extended by a closed-loop. In order to incorporate the feedback into the model and reflecting the results of the executed tests as well as the state of the test model, model reflection is needed: Model reflection describes the instantiation of a model at run time with the current state of the SuT, the environment, the test results, among other things. To enable that reflection and to use the information for test case generation and execution, or even for adaptation of the test model, run time models are needed. Run time models, following the definition of Aßmann et al. [12], are models that, first, can be instantiated with a concrete state and, second, are executable.

6.2.1 Feedback in Model-Based Testing

Model-Based Testing is, as introduced above, a static process for the automation of testing activities. This process and method have been proven, as shown by Binder et al. [18], to be very successful when established for systematic testing of complex systems. The information of the test engineer is set and, in an open-loop-control manner, the test activities are carried out without considering the output of the system. As described by Binder et al. [18], that works quite well, even for complex systems. However, testing SO mechanism extends the challenges for testing by the following:

- Making SO mechanisms testable as well as isolating and integrating SO mechanisms for testing
- Coping with error masking of SO mechanisms in testing
- Providing a test oracle for SO mechanisms

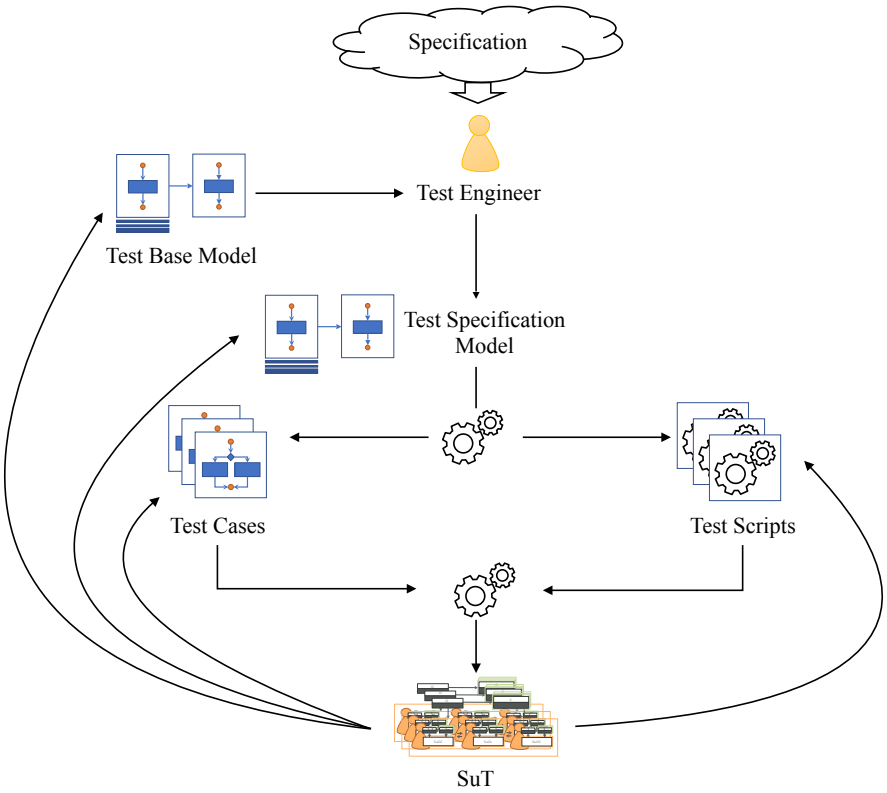


Figure 6.2. The generic MBT process from Figure 6.1 has been extended here by feedback. That is closing the loop between the SuT and the test model. Further, the test model itself is split into a test specification model and a test base model. The latter is delivering input to the test engineer by providing standard information used for MBT, the test specification is an extension of that model.

- Coping with a huge state space

These requirements are derived in Chapter 2. At a first glance, it looks as standard requirements of complex systems, however, the characteristics are entirely different to address. That is due to the fact that the systems are highly depending on the environment and are acting in a partially unpredictable way according to the environment. We have to cope with that by closing the loop of MBT. Figure 6.2 illustrates that idea, the information from the SuT as well as its test execution is fed back into the model, the test cases, and test scripts. Thus, this information can be used for controlling the test execution, the test generation, and the test case evaluation; consequently, for the whole test activities.

Establishing a closed-loop MBT requires the following:

1. A *monitor* of the SuT and its environment is needed to feed back the output into the test system.
2. A *processing* needs to be carried out, i.e., the test model instances need to be updated (including encoded test goals).
3. A *situation aware test case generation* is needed, that is producing the input to the SuT (in control theory the plant).
4. A *test goal* is needed to steer the test case generation.

The monitor is part of the test scaffolding, already described and discussed in Chapter 5, where different parts of the SuT are monitored in order to provide the necessary information. Which information is necessary is directly derived by the test model, where each part needs information about the current state by the monitor. This information is processed in the run time model, a concept that is described in Section 6.2.2, by using the concept of reflection, introduced in Section 6.2.3. The test case generation, that is described in Sections 6.3 and 6.4, is able to use the run time model and the provided information for test case generation, that is able to fulfill the test goals, discussed in Chapter 7 as well as throughout Sections 6.3 and 6.4. This forms the approach of MBT for SO mechanism, based on feedback.

6.2.2 Concept of Run Time Models

The model in MBT is in general consisting of three main components [68]:

1. The system model,
2. the test model, and
3. the environment model.

Whereas the system model is a mapping of the SuT, the test model is a description of the test cases for the SuT within the model, and the environment model describes the SuT's relevant environment. Within these models, it is possible to conceptually describe the SuT, its environment, as well as its test suite. Further, this model is used to reflect the current state of all these components during testing by merely instantiating the model. The current instance of the model should then reflect the current state. For this purpose, so-called run time models are used. These models are characterized by the ability to be instantiated at run time. Thus, the model can reflect the current state of

the modeled system in order to reason about its state. Besides this ability, the run time models used in this thesis are also executable. The execution of the model allows for the simulation of the modeled components. This simulation is used in this thesis for the test case generation: By simulating the test model possible inputs for the SuT are generated. Further, the executable model is used as a test scaffold. The run time model is either used as a test stub or as a test driver. The driver provides the necessary binding between the SuT and the run time model. The test stub provides information needed by the SuT for its execution, by directly simulating the stubbed component.

Aßmann et al. [12] *"perceive reflection, modeling and separation of concerns as the three main pillars to achieve models@run.time"*. We follow this thought of run time models. For the implementation of the run time model in the MBT approach we use object-oriented programming languages for describing our models. Thus, reflection, modeling, and separation of concerns, as well as the fact that they are executable, is already built-in. Indeed, only a subset of the programming language is needed for the desired modeling language. We build upon the concepts of S#, a modeling language designed by Habermaier and Leupolz [71, 72, 91] for modeling safety-critical systems. S# brings forward established software engineering principles, e.g., reflection and separation of concerns, and best practices to the modeling and analysis during all phases of development. It is an integrated approach for the systematic development of comprehensible, adequate, and modular models [72]. S# is a modular modeling language based on the C# programming language. Thus, it provides a component-oriented domain-specific language built on top of an object-oriented language. Similar concepts can be applied in Java, as we will demonstrate in the evaluation (in Section 6.6).

6.2.3 Model Reflection for Reflecting Changes in the System under Test

Establishing a situation awareness of the test system, e.g., for test case generation, as well as being able to execute the test oracle, as described in Chapter 3, on the test model, requires a processing of the monitored data from the system and its environment into the test model (to be precise into the system and environment model). Model reflection is responsible for that task within the proposed closed-loop MBT approach. Knowing its structure and being able to modify it, if necessary, is the ability of model reflection. Model reflection is similar to the reflection known from programming languages like C# or Java. As we rely on these two languages as a modelling language, we can use these concepts of reflection here, too. To apply reflection, we have to analyze the executed SuT. This is done via test drivers, as described in Chapter 5. The test driver is written in the object-oriented language of the model and is responsible for getting the information from the sensors in the SuT. This is done either invasive, by incorporating code into the SuT, or non-invasive, by using the given interfaces of the SuT. The test infrastructure, as described in Chapter 5 offers interfaces to access the system. Nevertheless, there is a domain-specific implementation effort needed to implement these interfaces and the test driver for the specific SuT, as we will show in the evaluation in Section 6.6. The information gained from the test driver is used to change the instances of the system and the environment model (and if needed even the test model, if test cases are adapted, as

shown in Chapter 7) and also the structure. Changing the structure of the model at run time is only possible due to the reflection capability of the used modeling language.

Generating a Snap Shot During Model Reflection In a distributed system the generation of a consistent snapshot is far from obvious due to the needed synchronization. We gather the information for a snapshot at every test step (a test step lasts for a maximum of 300ms) by sampling the information in a fixed order. The resulting time difference of the sample (with a maximum of 300ms) showed no impact on the overall results of our investigated case studies. This is due to the effect that configurations outside the CCB that are missed within this time slot are not of interest since they can be considered as a successful reconfiguration by the adaptation mechanisms if the configuration is inside the CCB again. Further, the systems have shown to be not as quick in changing its configuration within milliseconds. Thus, the snapshot generated is valid for our testing approach. That is one of the results that is further discussed in the evaluation (Section 6.6).

6.3 Probabilistic Models for a Continuous Self-Organization Mechanism

Continuous SO mechanisms are characterized by the environment they are designed for, that is formed by the continuously changing properties. The domain and solution space of the SO mechanisms is thus continuous. In this section, we will demonstrate how the environment and the tests are modeled with probabilistic models. The underlying idea is that a profile of the continuously changing environment is formed. This profile describes the delta of a change as well as the likelihood of such a delta to occur. The system, test, and environment models are designed to enable this description within the run time model.

6.3.1 System Model

Starting with the system model, we can reuse the domain model resulting from the requirement analysis, as described in Chapter 3. This model is refined by adding technical classes that are needed for the implementation of test the framework and removing clutter not needed for the implementation of the test system. For generating the initial system configuration of a test run, a description of the test configuration is needed. The intention is to provide the information necessary to automatically generate and also reasonably constraining the following setting within a test run:

- Set of agents of different types
- Initial state of the agents
- Agent groups and members
- Initial system structure
- Fully parametrized SOuT

The needed specification is further subdivided into the test system and the SOuT.

The system model specifies all vital information from the domain the SOuT belongs to and is used for generating the system configuration within the test suites. Further, the system model is provided that is enriched by constraints describing the CCB (cf. Chapter 3). Concerning our power plant case study, the constraints define valid partitionings, e.g., the maximum and the minimum number of power plants for each Autonomous Virtual Power Plant (AVPP) as well as the constraint that each power plant must be contained in exactly one AVPP. Further, the behavior of each agent type (e.g., wind turbines, solar panels, or biogas power plants) is defined by standard design documents. In our power plant case study we additionally have to define *groups of agents* which are influenced by similar environmental changes, such as wind turbines with a specific locality. Accordingly, a group of agents shares one EP, a stochastic model abstracting possible environmental changes (more details are given at the end of this subsection). The fact that an agent is in a specific group is not known to the SOuT and has no direct influence on the partitioning decisions of the SOuT. The members of a group of agents can be of different agent types.

We define how the environment influences certain types of agents so that a reduction of the test cases to realistic scenarios is possible. Specifying the scenarios and their likelihood is due to the test engineer based on the environment, i.e., what is the realistic surrounding of the SO mechanism. Having the influences' relation to the agents controlled by the SO mechanisms, i.e., in some sense the test input, these scenarios are transformed automatically in realistic test data. The realistic test data is necessary to increase the reliability of the test result. Indeed, describing the influences relation between the environment and the types of agents is rather complicated. At this step we rely on simplification and abstraction within the model to keep the approach scalable: We assume that the mapping of environmental influences to agent types (in its abstracted form) can be determined and defined a priori, i.e., we neglect that the influence might change over time or is non-deterministic. Based on this assumption, a mapping function is defined from the environment state (relevant to the concerning group) to states of the agents within the group. Thus, the influence might be described by mapping delta-values, describing the change of the current state according to the appearance of an environment state. The consequences of this assumption and simplification are on the one hand that the model is scalable within the approach and can be handled by the test engineer, but, on the other hand, that it might neglect some situations for testing. The evaluation results (cf. Section 6.6), however, showed that it is still possible to find different kinds of failures and without having evidence that a failure has been overlooked.

The information which is described in this model encompasses the specification used for generating the system configurations of the test runs. For this purpose, a domain description (in the form of an UML class diagram) is used that is enriched by constraints for co-domains for the classes and their states.

To recap, the system model must contain at least:

- Types of components

- Definition of possible initial states for the components
- Suitable ranges for the minimum and maximum number of components of a specific component type
- Constraints concerning groups of components (minimum and maximum number as well as size)
- Mapping of environmental influences to agent components
- Constraints concerning valid system structures, i.e., the relevant part of the CCB

Additional parameters for the test cases generation are a minimum and a maximum number of test cases within each generated test sequence and the number of test sequences that should be created for each test suite.

Since the model of the SuT depends on the application domain, the overall description of the model itself is quite generic and coarse. However, this generic description needs to be transferred into a specific model of the SuT for each application case, as described in the evaluation in Section 6.6.2.

Further, the system model specifies valid configurations for the SOuT. It is used to derive valid ranges for all relevant parameters, such as the algorithm's maximum run time. The selection of relevant parameters highly depends on the concrete algorithm (e.g., some algorithms allow to specify a maximum execution time, and some do not), the situations that should be covered by the test runs (e.g., some failures only occur if we give the algorithm enough time), and the domain knowledge (e.g., in some domains, the maximum run time is naturally bounded). Clearly, in contrast to the random testing approach, a suitable parametrization can be used for directed testing. This means that we can push the algorithm into interesting directions, e.g., to use specific functionality, which might reveal failures of the SuT.

The parameters of the SOuT and dependencies between the parameters are specified as constraints of the permitted setting of the parameters of the SOuT. The model is specified with variables for the parameter that are restricted by domains and relations between them. The resulting Constraint Satisfaction Problem (CSP) is solved by the input component of the framework for forming a system configuration within a test run. For an SOuT that is based on a particle swarm optimization algorithm to form organizational structures (like the PSOPP algorithm described in Section 6.6.2) a parameter of the algorithm is the number of particles to use, constrained in the model of the SOuT, for instance, by an upper and lower bound. Further, the number of particles is related to the constraints concerning the number of partitions to be formed. This relationship is incorporated into the CSP. The described CSP is the foundation for automatically generating valid settings for the SOuT. The second aspect of the description of the test setting is the model of the SOuT.

6.3.2 Environment and Test Model

Recalling our idea of isolated testing of SO mechanisms from Chapter 5, the environment and test model are describing environmental changes and influences. This abstraction is because of possible interferences with other SO mechanisms due to interleaved

feedback loops as well as the huge state space induced by the different possible states of the environment and the SO algorithm's non-deterministic behavior. We address the problem of a huge, flat-branching state space by providing stochastic models of the environment and can describe its influences on the SOuT by functions describing the environment's influence on the system that enables to decouple the SO mechanisms.[2]

As we do not assume that all agents controlled by the SO mechanism share the same environment (e.g., because of their geographical distribution) and are equally influenced by environmental conditions, we define these EPs and influence functions concerning a specific group of agents $\mathcal{G}$.[3] This abstraction allows us to deal with large state spaces even better.

For example, it is not necessary to consider the complete set of possible states of the environment $\mathcal{E} = \{(\mathsf{cloudy}, \mathsf{high\ price}), (\mathsf{rainy}, \mathsf{high\ price}), (\mathsf{sunny}, \mathsf{high\ price}), (\mathsf{cloudy}, \mathsf{low\ price}), (\mathsf{rainy}, \mathsf{low\ price}), (\mathsf{sunny}, \mathsf{low\ price})\}$ if we regard a group of solar power plants whose output mainly depends on the current weather conditions and is more or less independent of the current market price (a property that is defined in the mapping of environmental influences to agent types). Instead, for each group of agents $\mathcal{G}$, we map one or more states of the environment from the set $\mathcal{E}$ to a single so-called *relevant state* that describes the relevant parts of the environment's state for $\mathcal{G}$, i.e., those that have an influence on $\mathcal{G}$'s behavior. By gathering all relevant states, we obtain the entire set of relevant states $\mathcal{R}_\mathcal{G}$ for $\mathcal{G}$. In case of our group of solar power plants the states $(\mathsf{cloudy}, \mathsf{high\ price}), (\mathsf{cloudy}, \mathsf{low\ price}) \in \mathcal{E}$ are mapped to a state cloudy that becomes a member of $\mathcal{R}_\mathcal{G}$. In this example, $\mathcal{R}_\mathcal{G}$ is finally equivalent to the set of weather conditions $\{\mathsf{cloudy}, \mathsf{rainy}, \mathsf{sunny}\}$ considered in $\mathcal{E}$.

The identification of relevant states is supported by the mapping of environmental influences to agent types as described in the model of the system under test. Thus, if the environment state or a set of environment states corresponds to an environmental influence that is already mapped to an agent type of the concerning agent group, this state has to be included into the set of relevant states for this agent group. However, mapping the environment's states to relevant states of an agent group is—as the mapping of environmental influences to agent types in the model of the system under test—not generically solvable; in every application domain, this classification of relevance has to be explicitly made. Further, the relevant states are not limited to the mapping described for the environmental influences to agents' types, since, among other things, the other SO mechanisms are here also part of the environment (whereas the model of the SuT does not consider that).

The exemplified description above shows what the necessary steps are and how this mapping should be achieved. The mapping, in general, does not have to be disjoint but we expect it to be complete since only useful environment states should be included in $\mathcal{E}$, i.e., states that influence at least one of the agent groups.

[2] Note that the environment also covers, in this case, other SO mechanisms of the system.

[3] This technique of state reduction is performed according to the state abstraction principles that are well known in classical testing [121].

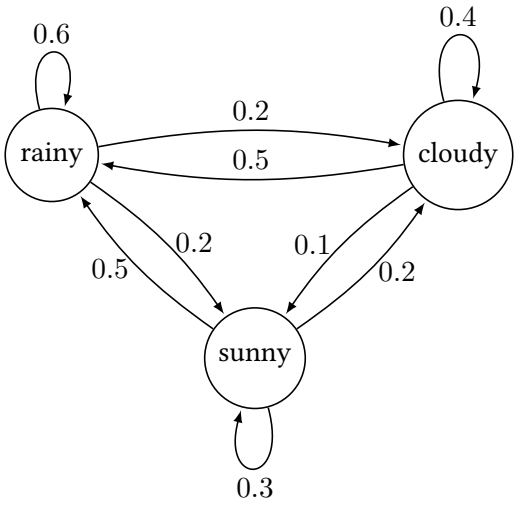


Figure 6.3. In this figure a transition graph model of an EP for a group of solar power plants at a designated location is shown. It shows that weather changes between "rainy", "sunny", and "cloudy" with a certain probability. These environment states are decisive for solar power plants, because they are mainly depending on the current weather condition.

Concerning a specific group of agents, an EP not only captures the relevant states $\mathcal{R}_\mathcal{G}$ of $\mathcal{G}$'s environment but also probabilities for changes from one state to another. Assuming that the next state only depends on the current state, an EP represents a first-order Markov chain. Figure 6.3 depicts a simplified example of an EP for a group of solar power plants in a specific region; as a matter of fact, the models used for testing are much more complicated (cf. Section 6.6.2). The transition graph model of the EP in Figure 6.3 is used to derive the model in Figure 6.4 where the states are refined by the description of state changes of the component group concerned. Such an EP can either be created by using domain knowledge or is derived from statistical data gathered during the execution of the SuT, or a combination of both.

To model the way the environment influences the members of an agent group $\mathcal{G}$, we use a function $f_\mathcal{G} : \mathcal{R}_\mathcal{G} \times \mathcal{S}_\mathcal{G} \to \mathcal{S}_\mathcal{G}$, where $\mathcal{S}_\mathcal{G}$ represents all possible states of $\mathcal{G}$'s members. With regard to a member $a \in \mathcal{G}$, the function $f_\mathcal{G}$ maps the new state $\sigma'_{env} \in \mathcal{R}_\mathcal{G}$ of $\mathcal{G}$'s environment and a's current state $\sigma_a \in \mathcal{S}_\mathcal{G}$ to a new state $\sigma'_a \in \mathcal{S}_\mathcal{G}$. For instance, the change of the current weather conditions from sunny to $\sigma'_{env} =$ rainy could impair a solar power plant's ability to make adequate predictions of its future output, which is reflected in the transition from $\sigma_a =$ good predictions to $\sigma'_a =$ bad predictions. Different influences of the weather on different types of weather-dependent power plants—represented as different agent groups—can be formalized by group-specific functions $f_\mathcal{G}$. On the other hand, if an EP describes possible developments of the prices at an energy market, $f_\mathcal{G}$ can model the way a power plant or consumer behaves at the market, i.e., its strategy. For example, if the market price falls below a certain threshold, some consumers might change their strategy to "buy energy", whereas the producers might become more reluctant to sell their production. As is the case with the creation of EPs, such influence functions can be deduced from domain knowledge and statistical

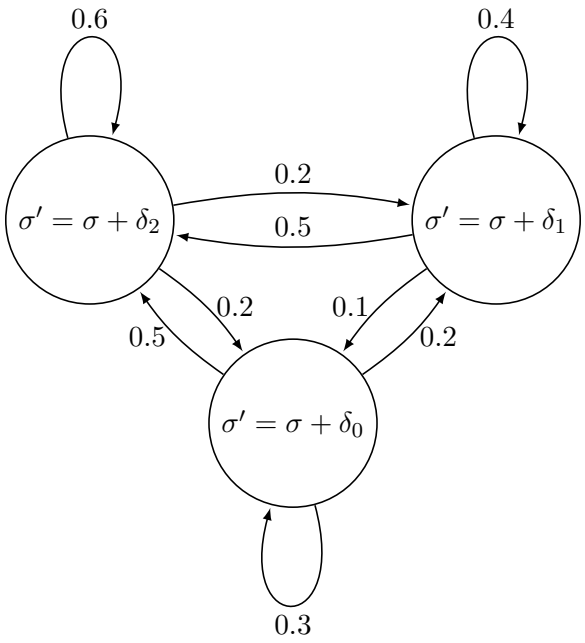


Figure 6.4. The transition graph model of the EP in Figure 6.3 is used to refine the model in this figure, where the states describe state changes of agents within the specified group. Every node in the transition graph model has some specific δ_i which describes the change between the current state σ of an agent which is within the context of the EP towards the following state σ'.

data. Note that, in some cases, the influence function might depend on random variables and thus becomes probabilistic. This reflects the fact that we cannot assume perfect knowledge about the influences of the environment on the agents due to the high complexity and the components' autonomy. However, it depends on the test designer's choice as well as on the application domain whether to use a probabilistic function to map the environment influences to the states of component groups. The drawback of modeling these functions on random variables is that the process of test case generation is less controllable, but it might reveal some interesting test cases. In the evaluation (cf. Section 6.6.2), we use deterministic functions which especially paid off in the process of getting more control over the test case generation procedure and gaining higher failure detection rates by incorporating domain knowledge of the test engineer. This reflects the overall goal of testing that is finding failures. Thus, it is worthy to abstract from some characteristics if that leads to better handling in test case generation procedure, in order to gain a higher failure detection rate. Of course, there might be applications where detailing the model applies better in order to gain a higher failure detection rate due to the more accurate model that is, for instance, able to reveal border cases that are failure prone. The choice depends here on the test designer that adapts the presented approach to specific algorithms and application domains.

As we will explain in the next subsection, the EPs are used for the test case generation by simulating the Markov chains, yielding random but representative test sequences describing environmental changes for the different groups of agents. The probabilistic

information of the EPs can be used to estimate the relevance of generated test cases and test sequences and their likelihood of occurrence in a realistic setting. As we will see in the evaluation in Section 6.6.2, this property makes EPs also a tool for coverage analysis.

In sum, the test model is used (1) to create a realistic test environment with relevant test sequences and (2) to identify relevant subsets of the complete state space.

To address (1) we introduced a probabilistic modeling concept where it is possible to reflect the uncertainty with a probability distribution according to the possible states. Within this concept, we introduced a context for the environment to handle the complexity of the environment model en bloc. The isolation of the SO mechanism is possible by using the technical concepts introduced in Chapter 5. These concepts are complemented by modeling not only the state changes of the environment but having a connection to the internal state of the component affected by an environmental change. This relation consequently unplugs a layer of indirection in testing the SO mechanism. Subsequent we introduced the techniques and concepts for shaping the ever-changing environment of SOAS within a test model.

6.4 Fault-based Testing Models for Discrete Self-Organization Mechanisms

Discrete SO mechanisms are designed for an environment where the change of environment properties (described in the environment model) is not a slight change, as described in the section before, but a discrete event for a change of a (mostly) boolean property. The production cell cases study is one example of a discrete environment: Tools of the robots are one part of that environment. They might either be broken or not. For the pill production, another example of discrete environment SOAS, it is the same; a conveyer belt either is working or not. The change in the environment is due to a discrete event. We are going to model these changes as environment faults since the event leads to an environment that does not match with the expectation of the system (a fault from the system's view) and demands for a reconfiguration. This notion is used in the test model, i.e., the model describing abstracts test cases for the SOuT. Letting a particular tool of a particular robot fail is a concrete test case for the SO mechanism since its responsibility is to maintain the ability of the production cell to fulfill its task under ever-changing conditions, the failing tool is one of such changing conditions (a discrete change to be precise). Having abstract test cases described as environment faults is on the one hand quite intuitive for discrete, boolean changes of properties and on the other hand common practice in testing. However, classical fault-based testing, as described by Morell [106], is in some sort different, but with the same intention: Fault-based testing is aiming at revealing different, specified faults within the SuT by directing the test efforts toward finding these faults. We are using the faults as test input, as the SO mechanism is dependent on the environment changes. Indeed, we also assume that by activating these faults, i.e., simulating the fault via the test driver (integrated into the test model), failures of the system should be revealed. Thus, we have introduced another dependency layer and a different interpretation for fault-based

testing. As our modeling language S# is incorporating the concept of faults and fault activation, we build upon these concepts presented by Habermaier [71]. The test model is used to describe the possible environment faults. These faults are completing the environment and system model.

6.4.1 The System Model for Discrete Self-Organization Mechanism

Describing the system test model for discrete SO mechanisms is quite similar to the continuous, as described in the section before (Section 6.3). The system model specifies the interfaces of the SO mechanism, as described in Chapter 5 in Figure 5.2 divided into the following components: the detection component, the computation, and the distribution component. That is exemplified in Listing 6.1 where a simplified model of the whole SO mechanism in the Hadoop case study is shown. The test engineer implements the interface of the SO mechanism component (cf. Figure 5.2) by defining the methods for the three SO mechanism functions. Depending on the integration level either a test driver or a stub is implemented in the function. The first is shown for the detection and the computation component. The methods call a function that maps to the SuT. The latter is the case for deployment, here just the model is updated, and the input is accepted from the SuT. The technical details are not shown in Listing 6.1, they encompass the system-specific implementation for connecting the SuT that are further discussed in Section 6.6.

6.4.2 The Environment and Test Model for Discrete Self-Organization Mechanisms

The environment and test models for discrete SO mechanisms are illustrated by the simplified model of the web-service system case study in Listings 6.2 and 6.3. Two different parts are of interest:

1. The description of the environment and its relation to the SOuT in the environment model.
2. The possible environmental faults and their behavior in the test model.

Both models are integrated into the description in one class per environment object. The first is described as shown in Listing 6.2 The properties describe the necessary information for different states (instances) of the environment object as well as its relations to the SOuT and the methods allow for manipulating the state and/or providing stubbing for the SOuT. The second part is exemplified in Listing 6.3, it extends the Listing 6.2 by faults. The faults are replacing merely the functionality stubbed in the environment with the functionality provided in the fault. We have two different kinds of faults. The first is persistent, i.e., the fault is replacing the function permanently, depicting that these kinds of environment faults are not repairable. The second kind is transient, i.e., the fault is present by activation and inactive afterward in a non-deterministic way. That simulates a highly unreliable environment. Indeed, there might be some more complex behavior where the faults take longer to be repaired, for that reason the `Fault` class might be extended with another fault behavior.

```
public class Controller : ISOMechanism
{
    public List<Client> ConnectedClients { get; private set; }
    public List<Server> ConnectedServers { get; private set; }
    public List<App> Apps { get; private set; }
      /* ... */

      public void Monitor()
       {
           MonitorServers();
           MonitorApps();
       }

    /* ... */

    public void ComputeSolution()
        {
            var nodeAllocation = ComputeServerAllocation();
            /* ... */
            DeploySolution(nodeAllocation);
        }

    public void DeploySolution(Dictionary<Server,ServerAllocation> serverAllocation)
        {
            foreach (Server server in serverAllocation)
            {
                server.updateAllocation(serverAllocation[node]);
            }

        }

    /* ... */

}
```

Listing 6.1. The `Controller` is the SO mechanism in the web-service system case study, responsible for reorganizing the `Servers`. The simplified version of the model shows a set of properties forming the system model of the SO mechanism. The `ISOMechanism` interface is part of the test base model that is defining the three parts of an SO mechanism, that are implemented in `Monitor`, `ComputeSolution`, and `DeploySolution` here. These methods are either test drivers in the implementation or stubs. `Monitor` and `ComputeSolution` are showing a driver calling the driver methods that needs to be implemented and call the SuT. The `DeploySolution` is a stub that distributes the solution in the model.

```
class Server : Component {

  Controller _connectedController;
  bool _isActive;
  List<Query> _executingQueries;

  public void Activate()    {
      _isServerActive = true;
  }

  public virtual void AddQueries(List<Query> queriesToExecute) {
    _executingQueries.AddRange(queriesToExecute);
  }

  public virtual void UpdateAllocation(ServerAllocation serverAllocation)
  {
      _isActive = serverAllocation.isActive;
      _connectedController = serverAllocation.conntectedController;
      /* ... */
  }

  /* ... */

}
```

Listing 6.2. Simplified S# component representing a web-service `Server`. The `Server` is mainly described by its properties `_isActive` and `_connectedController`. The first is describing its abstract status and the latter one refers to another model element, the controller. The provided methods allow for stubbing as well as updating the model state.

6.4.3 Designing Test Models with Environment Faults

Technically, environment faults can be added to every environment object that is showing some behavior. There are different types of environment faults: the environment fault may cause no behavior at all, a faulty behavior leading to a wrong state at once or after a few time steps, amongst others. The concept of changing the environment with a faulty behavior is related to the well-known fault-based or mutation-based testing approach, proposed by de Millo et al. [43]. Mutation testing assumes that errors in programming are not simple (the competent programmer hypothesis) but of a similar kind. Thus, if a test suite finds one of this kind others will be found, too. For that purpose, errors are added to the software, for different kinds and a test suite is optimized to find these, the errors are removed afterward, and the test suite is re-run on the SuT (and reveals actual failures). In the case of testing SO mechanisms, where the structure of the program itself highly depends on the environment, the structure and state of the environment are decisive for the execution of the SO mechanisms. That is why we are focused on modeling this environment as a test input for the SO mechanism. We mutate the environment with the environment faults. Thus, similar rules for mutations apply to create environment faults. However, it is not the resilience of the test suite which is evaluated. It is the resilience of the SO mechanism in order to reveal failures. The information which faults are relevant for the test model and which fault behavior is chosen is due to the test engineer. The decision is focused on the tested behavior of the SO mechanism. This is done by concentrating on aspects of the environment that are

```
class Server : Component {

  Controller _connectedController;
  bool _isActive;
  List<Query> _executingQueries;

  public void Activate()    {
      _isServerActive = true;
  }

  public virtual void AddQueries(List<Query> queriesToExecute) {
    _executingQueries.AddRange(queriesToExecute);
  }

  public virtual void UpdateAllocation(ServerAllocation serverAllocation)
  {
      _isActive = serverAllocation.isActive;
      _connectedController = serverAllocation.conntectedController;
      /* ... */
  }

  [Transient]
  class ServerCannotActivate : Fault {
    public override void Activate() { }
  }

  [Persistent]
  class CannotExecuteQueries : Fault {
    public override void AddQueries(List<Query> queriesToExecute) { }
  }

  /* ... */

}
```

Listing 6.3. Simplified S# component from Listing 6.2 extended with faults. The `Transient` annotation indicates a fault that may be presented and repaired afterward in a non-deterministic way, the `Persistent` annotated fault is active for the rest of the execution, as the functionality might not be restored.

described in the goal model and the derived CCB, as described in Chapter 3. The included faults should cover at least all properties that are part of the CCB and be faulty in a way to execute different kinds of SO that are needed. The standard mutation operators could be used to define a different kind of faults. The mutation operators [4, 19, 94] that apply for design environment faults are the follwoing:

- Information about the object's state is not provided
- Information about the object's state is provided in a wrong way (type, format, content, order)
- Information about the object's state is processed wrong (extended or limited)
- Information about the object's state is provided too late/early
- Object is blocking information or critical sections in the execution

These operators are used to provide standard operators for environment faults. The operators have to be checked if they modify the behavior in a way that the SO mechanism is effected, that is done by checking the CCB constraints and domains.

6.5 Back-to-Back Testing of Test Model and Implementation

Engineering the overall test model proposed for SO mechanisms in this chapter is challenging. Many decisions and assumptions have to be made based on the requirements of the SuT. The environment needs to be adequately defined and described, either by probabilistic or fault-based models. The system components have to be identified and assigned with properties. Further, the CCB needs to be designed and derived, as described in Chapter 3, in order to build the test oracle and use it as a foundation of the aforementioned tasks. This design demands expertise and good modeling skills from the test engineer. Indeed, the development engineer needs to perform a similar task in developing the actual system. This effort is also crafted from the set of initial requirements. In both tasks, human errors by the engineers have to be expected. In order to support these hard engineering tasks, we follow the BtB testing concept, proposed by Vouk [166]. The primary virtues of this method are a structured process to automatically detect differences in the understanding of requirements, or even erroneous requirements, through deviating interpretations necessary for implementation. Such deviations can be used for debugging and discussing whether the test model or the implementation is correct and needs correction and/or refinement. This form of quality assurance increases users' trust in the obtained models and is indispensable for the practical usage of the overall approach.

In a nutshell, the approach relies on the interplay of a test engineer with a development engineer that work together to formalize informal requirements properly. The development engineer is responsible for an efficient and correct SO mechanism implementation. The test engineer has to provide a correct test model as well as the necessary connection of the SuT with the test framework. Upon presenting a test input to the SuT and the test model, we can at least identify deviating opinions of, e.g., whether a situation is valid or not. For the development engineer, this decision involves selecting the right type of

reconfiguration(s) to match the requirement. This feedback leads to modifications of either of the test model, the SOAS, or the requirement's specification.

6.5.1 Using Executable Run Time Models for Back-to-Back Testing

The approach for BtB testing is making use of the concepts presented in previous sections of this chapter and throughout this thesis. The test engineer uses the requirements to build the CCB, as described in Chapter 3, the SOuT is defined within the test setting by its interfaces, the decomposition is designed as described throughout Chapters 4 and 5, and the corresponding test model is designed as described in this chapter. All these steps of the development of a test setting are needed for BtB testing. Further, the SOuT has to be connected to the test framework in order to execute the tests. As the test model is an executable model, we are able to start BtB testing directly. Having an executable model allows for executing and generating input on the test model as well as executing the input on the SuT and comparing the results by mapping the state of the SuT into the system model and checking it with the test oracle. These are the necessary prerequisites for BtB testing, coming right out of the box. In Chapter 5, we discussed the disassembling and isolation as well as the stubbing of the SO mechanism. Using these concepts allows for applying BtB testing at different development stages. It is possible to BtB test the SO algorithm, the observer part, or the deployment SO mechanism. However, the part to be stubbed needs to be implemented in the system model to be executed.

As the executable models are object-oriented code, a further possibility is to use debugging along with BtB testing, i.e., the execution of the model could be paused and in-depth investigated according to its state. That allows for identifying the mismatch in detail, saying the model operation that is the first differing from the SuT can be identified.

For the time being, the execution of the test model is random, i.e., random environment faults are selected, and random transitions in the EP are taken. Different test case selection techniques are discussed in the next chapter.

6.5.2 The Special Case of Back-to-Back Testing Self-Organization Mechanisms

Self-Organization mechanisms are responsible for adapting the structure of a controlled system to an ever-changing environmental surrounding. The test model, presented in this chapter, can model this highly volatile environment of the SOuT. The adaptation of the SOAS is made possible by, what we called, underspecification (cf. Chapter 3), i.e., the system is allowed for decision making at run time as long as it stays inside the CCB or returns into the CCB if possible. Designing the CCB from an initial set of requirements and implementing the SO mechanisms within the CEI is based on decisions according to the range of the SO. The development engineer has a scope of discretion how wide the CCB is, figuratively speaking. The same task is due to the test engineer, who has to design and derive the CCB, too, in order to implement the test framework. This is different from systems with a precise specification where every interaction with the system is defined by an expected behavior. However, even in these

systems, Vouk [166] showed that there is often a mismatch between the implementation of the requirements and the test engineer's perspective on it. That is why BtB testing is used for these systems. The particular case of how the behavior of SO mechanisms is specified leads to the fact that BtB testing is even more needed here. This is mainly since SO mechanisms are more loosely specified, as elaborated in Chapter 3, is giving more room for interpretation. Thus, the interpretation of the expected behavior of SO mechanisms has been investigated in an BtB testing process, as proposed in this section.

6.6 Evaluation

We will show different aspects of the so far proposed approach for MBT for SO mechanism evaluated on different systems. The systems used for the evaluation are described in Chapter 2 in detail. We will complete, if necessary, that description here with more insights on the investigated approaches. The evaluation for the concepts of MBT includes not only the aspects presented in this chapter, but also the aspects of specification of SOAS and deriving a test oracle (Chapter 3, using the concepts of the CEI and the concepts for isolation and integration presented in Chapters 4 and 5). These concepts are all needed to establish testing, however, till the presented concepts of MBT we were not able to test the system. Each case study has different aspects that we use to demonstrate the approach proposed in this thesis. Thus, not every aspect of the testing concept will be described for each case study. The focus is on demonstrating the abilities of the approach, and thus we choose the case study accordingly for this purpose.

In this section, we will investigate the following aspects:

R1 Is the way of specifying SO mechanisms with the CCB applicable for different kinds of SO mechanisms?

R2 Is it possible to derived test requirements based on the CCB and define a test model?

R3 Is the CEI testable, i.e., observable and controlable, as proposed for testing?

R4 Is the proposed approach for isolating and disassembling the SO mechanisms applicable for testing?

R5 Is the proposed approach for integrating the SO mechanisms applicable for testing?

R6 Does the proposed test architecture enable testing SO mechanisms?

R7 Is the closed-loop MBT approach for continuous SO mechanisms able to execute an SOuT and reveal failures?

R8 Is the closed-loop MBT approach for discrete SO mechanisms able to execute an SOuT and reveal failures?

R9 Is the BtB testing approach able to support the engineering process by revealing failures in the test system and the SOuT?

R10 Does the MBT approach enable to build generalizable test concepts and models for system classes?

R11 Besides full integration testing, is it possible to address system testing within the given concepts?

R12 Is the approach scaleable for industry-size systems?

R13 How usable are the presented concepts for test engineers new to testing SO mechanisms?

In order to supply answers to these questions, we will investigate the following aspects of the implementation of the case studies (presented in Chapter 2).

R1 to R9: First, two case studies, one of the continuous and one of the discrete class, are investigated thoroughly from the specification of the CCB, defining the test models, disassembling the SOuT, integrating it in the proposed test scaffold and executing tests on it. This will guide us through the questions *R1* to *R9*. The two case studies are the production cell and the energy grid. The production cell has a central SO mechanism that is controlling a complex environment that is discrete. We will focus here on testing on the integration level where the SO mechanism is completely integrated. One interesting aspect here is that the SO mechanism and the test framework have been developed in a real BtB setting: two different engineers implemented it at the same time BtB. We will discuss these aspects. For the full disassembling of an SO mechanism, we will extract the SO mechanism from the energy grid. Here, the disassembling is highly demanding as the SO mechanism is regio-central, i.e., there are distributed blocks which are autarkically organized by an SO mechanism. Further, this system's SO mechanism controls a continuous environment. These two investigations are able to cover the questions *R1* to *R9* fully.

R10 to R13: We will further take a look at the other three cases studies where we are investing questions *R10* to *R13*. The implementation of the load-balancing web service has been developed in a student project accordingly with a test framework as proposed in here, supplying insights to the questions *R13*. The case study of the pill production will focus on the aspect of abstraction and reusability of the test models, focusing on question *R10*. The pill production is sharing the same underlying paradigm of the resource-flow with the production system, even though it has been invented and implemented by different research groups. We extracted the generic concepts of the test model for resource-flow systems and will show the abilities of the MBT approach for generalization. For the Hadoop case study, we selected an application from industry to show how the approach is scaling in this setting (*R12*). Further, we are going to test this system on a system-level, going beyond the integration of the SO mechanism and supply an answer to *R11*, too. We will show how that is possible and investigate the abilities of the approach for scaffolding in depth.

6.6.1 Production Cell—Testing an Integrated, Discrete Self-Organization Mechanisms in a Back-to-Back Test Setting

The production cell is a case study introduced in Chapter 2. The implementation and concepts for SO are the results of a phase of six years of intense research and development in an DFG founded program, resulting in two dissertations [109, 144]. We classified the system and its SO mechanism as discrete in Chapter 2.

Within this section, we will address *R1-R6* and *R8-R9*. Despite the fact, that the system has been already implemented, the SO mechanism has been newly built in a team consisting of one development engineer and one test engineer for this evaluation. The requirements, formulated in [109, 144], have been used independently for the development of test system and SO mechanism. The following discussion of that development will focus on the test system. However, the feedback from the development engineer is also incorporated here.

Development of the Self-Organization Mechanism

In the case at hand, we used MiniZinc[4] as a constraint modeling for designing an SO algorithm that can allocate the roles in the production cell. The SO mechanism is consequently a central solution, that can access information from all system components and also control them.

The system requirements have been translated into constraints formulated in a MiniZinc model that describes valid configurations for the production cell; an exemplary MiniZinc input for a system configuration is shown in Listing 6.4. Thus, it is possible to feed the SO algorithm with a specification of a task, the number of agents (carts and robots), the capabilities, and the routing table. If satisfiable, the SO algorithm returns a solution that assigns each tool needed for the task to some robot and that routes the carts between the robots accordingly. To complete the SO mechanism an observer has been developed in Java which is mainly consisting of boolean functions to be called at each time step of the system to check the current system configuration. That monitor is integrated into the Jadex implementation of the original project. Further, the distribution component is an adapter between the MiniZinc results and the Jadex agent system in order to update the role allocation.

Scaffolding for the Self-Organization Mechanism

This SO algorithm has been plugged into S# via an interface that provides the specification of the problem to be solved by the SO algorithm and that parses the MiniZinc results. That includes generating a text-input file for MiniZinc and converting a text-output file into an instance of the test model. An exemplary file is shown in Listing 6.4, the file format is input as well as output, showing the current configuration or the configuration to be. The constraints of the observer of the SO mechanism—originally developed in *Java* for our implementation of the production cell based on the multi-agent system

[4] `http://www.minizinc.org/`

```
task = [1,2,3,4,5,6]; noAgents = 6;
capabilities = [{1},{3},{4,5,4,2},{5,6},{},{}];
isConnected = [|true,false,false,false,true,false
                |false,true,false,false,true,false
                |false,false,true,false,false,true
                |false,false,false,true,true,true
                |true,true,false,true,true,false
                |false,false,true,true,false,true|]
```

Listing 6.4. The input model for MiniZinc describing a task, the available capabilities of the robots, as well as the connection matrix based on the carts' routes, corresponding to the configuration instantiated by Listing 6.7.

Jadex[5]—have been manually converted to C# in order to integrate them into the S# model. Indeed, a cross-compiler from Java to C# could also have been used. However, there were only slight syntactical adaptations necessary. Thus, the conversion was done manually. Indeed, every revealed failure was also checked whether this conversion had caused it, that was not the case for this evaluation.

Development of the Corridor of Correct Behavior and Deriving the Test Oracle

The development starts with the main system goal from the requirements. The primary system goal of the production cell is to process the workpieces according to their task. During requirements analysis, this goal is refined to the system goals visible on the right-hand side of Figure 6.5: the resource has to be processed according to a role allocation, which defines which agent performs which capability and how the carts transport the resources. Further, an obstacle has been identified: the role allocation can become invalid due to environmental changes such as a capability that is no longer available. This circumstance is captured in a hierarchy of obstacles, shown as red rhombuses.

A new goal is introduced to change a role allocation if necessary, to mitigate this uncertainty. Refinement of this goal leads to a requirement for the agent: the capability it has to apply must be available to it. Formally, this can be captured in the Object Constraint Language (OCL) as:

context Agent **inv** capabilityConsistency:
self.availableCapabilities
→ **includesAll**(self.allocatedRoles.capabilitiesToApply)

The basis for this formulation is the domain model of a self-organizing resource-flow system depicted in Figure 2.4 in Chapter 2.

The *Capability-Consistency* constraint is by far not the only one that needs to be observed. The requirement "Agent communicates the loss of a capability it is configured to apply" could be further refined to yield a requirement that observes neighboring agents by sending them heartbeat messages. If an agent does not answer anymore, the *I/O-Consistency* constraint is violated that states that agents' with which resources are exchanged have to be reachable. This constraint, as well as the part of the invariant should be observed accordingly by the test oracle. As already discussed in Chapters 3 and 4, the test oracle further needs to know whether or not there exists a valid solution

[5] http://www.activecomponents.org/

to be found and deployed by the SO mechanism. The algorithmic of this function is shown in Algorithm 1.

The result is a complete KAOS model that is shown in parts in Figure 6.5, an extended domain model that is including all the monitoring infrastructure for the test oracle, shown in parts in Figure 6.6, and OCL constraints that are included in the KAOS model that are transformed into C#, like the one shown in Listing 6.5.

Require: robotAgents, cartAgents, tasks
Ensure: a Boolean value indicating whether a reconfiguration is possible
1: $m \leftarrow$ GetConnectionMatrix(robotAgents) // transitive closure of all connected robots
2: **for all** $t \in$ tasks **do**
3: **if** $\neg\forall c \in t$.Capabilities: $\exists a \in$robotAgents: $c \in a$.AvailableCapabilities **then**
4: **return** false
5: **end if**
6: $A \leftarrow \{a \in$ robotAgents $\mid t$.Capabilities[0] $\in a$.AvailableCapabilities $\}$
7: **for** $i = 0$ to $|$task.Capabilities$| - 1$ **do**
8: $A \leftarrow \{a \in m[a'] \mid a' \in A \wedge t$.Capabilities$[i+1] \in a$.AvailableCapabilities $\}$
9: **if** $|A| = 0$ **then**
10: **return** false
11: **end if**
12: **end for**
13: **end for**
14: **return** true

Algorithm 1. Checks whether a reconfiguration is possible for a given set of robot and cart agents as well as the tasks to be carried out.

Building the Test Model

The model in the here proposed MBT approach consists of a system, an environment, and a test model. For the system and the environment model we use the classes from the domain model of the KAOS model, as shown in Figure 6.6, to build the S# model by defining the classes and the properties. This is done from the already discussed information. The test model is based on environment faults, Listing 6.6 shows an excerpt of the test model written in S#. These are two of twelve environment faults that have been specified for the robot and the carts, the two environment classes of concern.

As shown in Listing 6.7, we can build different setups for the testing. We call this virtual commissioning for testing SO mechanisms.

Virtual Commissioning of SOAS Systems The concept of virtual commissioning is mainly applied in the field of large manufacturing systems where a virtual manufacturing system is built in order to simulate individual manufacturing processes for optimization and validation purposes [89]. Within this virtual environment, the real

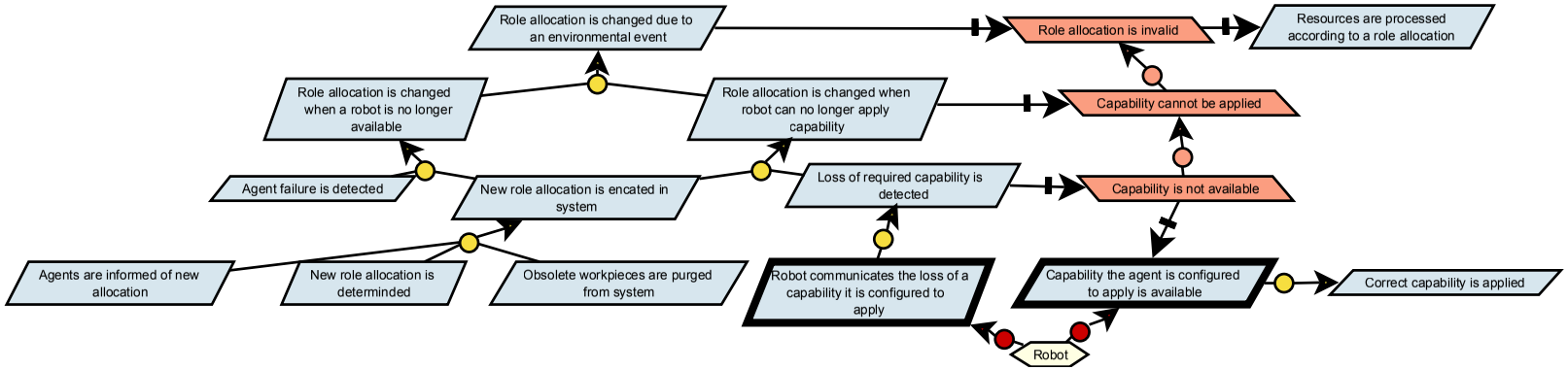


Figure 6.5. Showing an excerpt of the Knowledge Acquisition in Automated Specification (KAOS) model representing the requirements for the production cell. It is shown how a high-level goal, the Role allocation is changed due to an environmental event is broken down and mitigated to introduce self-organization. At the leaves two requirements, i.e., goals that could be assigned to an agent (a component in the system in KAOS speak) that is responsible for its fulfillment, are depicted. These requirements are formulated as OCL constraints and are transformed into a test oracle, as shown in Listing 6.5.

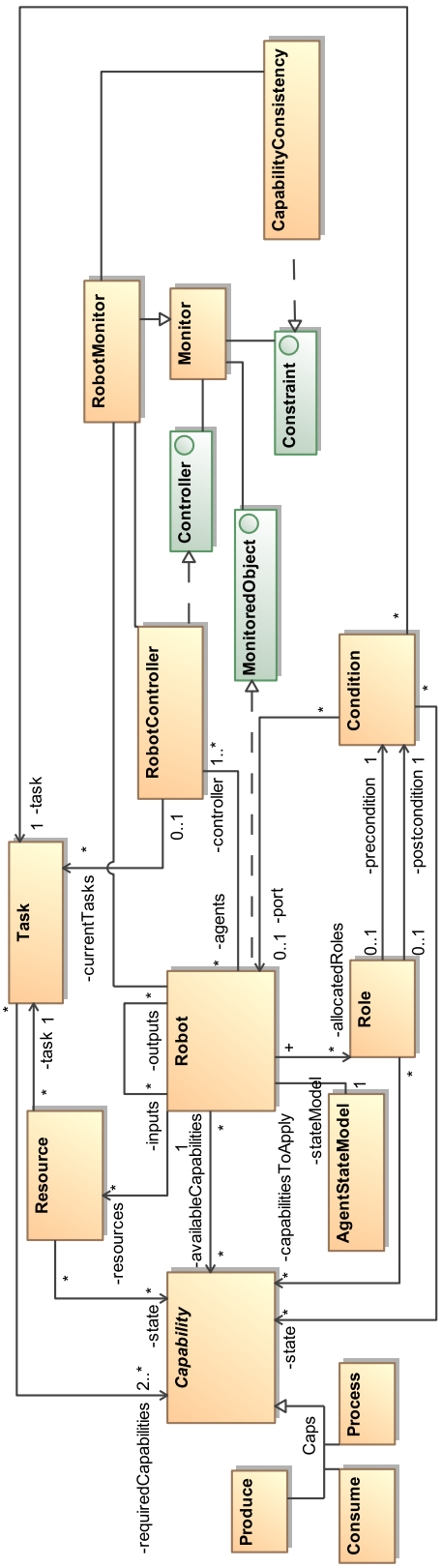


Figure 6.6. An excerpt of the transformed system model is shown, that has been completed by the monitoring infrastructure for the test oracle. That model is used as the foundation of the system and the environment model of this case study. Further, it serves as the domain model for formulating the constraints and selecting the agents in the KAOS model, cf. Figure 6.5.

```
public static IEnumerable<ITask> CapabilityConsistency(BaseAgent agent)
    {
        yield return RoleInvariant(
            agent,
            role => role.CapabilitiesToApply.All(cap =>
                        agent.AvailableCapabilities.Contains(cap))
        );
        /* ... */
    }
}
```

Listing 6.5. Showing a extracted part of the test oracle, here the capability consistency constraint has been transformed into C# code, enabling executing it on the S# model we use for testing.

```
        /* ... */

        public readonly Fault Broken = new TransientFault();
        public readonly Fault ResourceTransportFault = new TransientFault();

        /* ... */

        [FaultEffect]
        public class BrokenEffect : RobotAgent
        {
            protected override bool ApplyCurrentCapability() => false;
            protected override bool CanApply(ProcessCapability capability) => false;
            protected override bool TakeResource(Cart cart) => false;
            protected override bool PlaceResource(Cart cart) => false;

            protected override bool CheckInput(Agent agent) => false;
            protected override bool CheckOutput(Agent agent) => false;

            public override IEnumerable<ICapability> AvailableCapabilities =>
                Enumerable.Empty<ICapability>();
        }

        [FaultEffect]
        public class ResourceTransportEffect : RobotAgent
        {
            protected override bool TakeResource(Cart cart) => false;
            protected override bool PlaceResource(Cart cart) => false;

            protected override bool CheckInput(Agent agent) => false;
            protected override bool CheckOutput(Agent agent) => false;
        }

        /* ... */
```

Listing 6.6. Showing an extracted part of the S# test model where the faults are defined. They are simply overwriting the functionality of the environment that is mapped into the environment model.

```
CreateWorkpieces(5, produce(), drill(), insert(), tighten(), polish(), consume());
CreateRobot(produce());
CreateRobot(insert());
CreateRobot(tighten(), polish(), tighten(), drill());
CreateRobot(polish(), consume());
CreateCart(new Route(Robots[0], Robots[1]), new Route(Robots[0], Robots[3]));
CreateCart(new Route(Robots[2], Robots[3]));
```

Listing 6.7. Parts of the S# instantiation code for a configuration of the case study consisting of five workpieces that require the task produce, drill, insert, tighten, polish, and consume to be carried out on them. Four robots are created with some minor redundancy in available capabilities. The two carts connect all four robots via bidirectional routes.

controller is executed on the virtual plant enabling to test, tune, or initialize it for a specific configuration of the plant. We adopt this concept for the reduction of possible configurations of the system to be tested. The idea is to base the tests on only one configuration, namely, the one which should be rolled out afterward. Indeed, there will be changes at run time, e.g., new robots are integrated, new tools are added, or tasks change. Before such a change is rolled out to the running system, the model instance must first be updated, and the tests have to be re-run on the new instance. Since the change of the current configuration of the system is due to a human intervention—we assume the system is not able to extend itself by other components or similar—it is possible to run this test-first-deploy-after strategy while the real system is running separately. Thus, we select only the configuration for testing that is crucial for the deployment and have the ability to test new configurations on demand. This is possible due to the generic S# test model in which it is easy to instantiate new configurations (cf. Listing 6.7) and to automate the testing process.

This forms altogether the test model for the production cell.

Discussion of the Testing Results

For evaluation purposes, we analyzed different configurations (cf. Table 6.1) of the production cell. The configurations differ in the number of agents (robots and carts), the average number of capabilities per robot, the number of tasks, and the number of routes established by the carts between the robots. The test case generation technique at this stage is simple random testing, i.e., the faults are activated in a random order for testing. Further concepts for test case generation and selection are discussed in Chapter 7.

One main achievement of the evaluation is that it was possible to reveal the following faults[6] with the implementation of the SO mechanism; each fault is annotated with the responsibility of the SO mechanism, as described in Chapter 4, where the fault was detected:

[6] Note that the description is emphasizing the fault, i.e., the incorrect state of the SuT due to a human error, and not the observable failure. Thus, it is possible to give more insights into the faulty behavior and describe the root cause.

#robots	#carts	#capabilities per Robot	#capabilities per Task	#routes	#test cases	time (in min)
4	3	2.75	6	6	131,000	570
3	2	1.67	5	4	49	0.2
3	2	3.67	5	4	26,763	69.25
3	2	1.67	5	6	157	0.78
3	2	1.67	8	4	47	0.38
5	2	1.6	5	5	1,577	6.88
3	4	1.67	5	5	369	1.08

Table 6.1. Statistical data concerning the configuration used in the evaluation, the number of test cases generated and executed, the required time. Note that the time is used for complete testing. Note that the runs within our framework are deterministic, i.e., there is no need to consider mean values or standard derivations.

F1 The fault affected route handling: the MiniZinc implementation interpreted transitive routes as direct ones. Its computed configurations included direct connections that were not available, e.g. $0 \rightarrow 2 \neq 0 \rightarrow 1 \rightarrow 2$ (Computation).

F2 The fault was that the SO algorithm expected the routes to be unidirectional while they were bidirectional. The failure manifested itself as overlooked solutions even though at least one existed (Computation).

F3 The fault was a wrong implementation of the interface for the SO algorithm. The interface expected first the capability of a designated agent but got the first capability of the task assigned to the designated agent (Computation).

F4 The fault was a wrong format for the mapping of the solution from the SO algorithm to the system model concerning the pre- and postconditions of a role (Distribution). The pre-/postconditions contained the state of the workpiece in form of the remaining part of the task, e.g., for task $[D, I, T]$ the precondition contained $[D, I, T]$ and the postcondition $[I, T]$ if D had been performed. However, the mapping should lead to states of the workpiece representing the part of the task which already had been done, e.g., for task $[D, I, T]$ the precondition should contain [] and the postcondition $[D]$ if D had been performed (Distribution). This fault was detected even though the testing approach was initially not focused on Distribution.

F5 The fault was a too narrow restriction in the SO algorithm that did not allow to use intermediate robots that apply no tools since the maximum length of concatenated roles was restricted. Thus, Listing 6.4 was mistakenly considered to be *unsatisfiable* instead of returning the following solution, for instance: agents = $[1, 5, 4, 6, 3, 6, 4, 5, 2, 5, 4, 6, 3, 3, 6, 4]$; workedOn = $[1, 0, 0, 0, 2, 0, 0, 0, 3, 0, 0, 0, 4, 5, 0, 6]$ (Computation).

F6 The fault was a missing constraint with the observer, namely the I/O-Consistency constraint checked in the oracle. The failures occurred after activating a component fault that deactivates a cart that is part of the active task (Detection).

The faults F3, F4, and F6 have been detected in all investigated configurations. Indeed, F1, F2, and F5 mainly depend on the routing structure used in the configuration, e.g.,

smaller configurations would not be able to reveal the faults. F6 mainly depends on changing the active robots or carts of a task, since their removal might not be detected and the controller is consequently not activated. All detected faults mainly concern misinterpretation of requirement specifications. The kind of faults that we detected underpins one of the strengths of the BtB testing approach: the ability to reveal faults which are the result of a misinterpretation of the specification.

Discussion of the Research Question *R1-R6* and *R8-R9* Having discussed the evaluation that has been carried out within the production cell scenario, we will now elaborate on the research questions. The test setting, which has been implemented in an BtB testing endeavor is grounded on the systematic derivation of the CCB. We showed how this is done in the described case study. The result was a quite huge KAOS model resulting in nine different constraints of different complexity. These constraints are assigned to the agents of the domain model and were sufficient for deriving a valid test oracle. In the case at hand, we can compare the outcome not only with the developer's view but also with the previous work documented in [109, 144] that also follows the CCB concept. The results were comparable and covered the same requirements for the CCB. The overall effort for compiling the documents was about three to four person-days (plus becoming acquainted with the case study) and is rather low for a complex system. Having the results of the model at hand is very useful for starting the MBT approach, as presented in this chapter since the documents are directly used to build the first version of the model. Indeed, this model needed some refinement during the development of the test model and needed to be connected to the SuT, but it turned out to be an excellent input here. Concluding with *R1* and *R2*, the specification of the SO mechanism with the KAOS model and deriving the CCB within this approach by applying the relaxation of the goals worked well on this complex case study and are a well-suited input for the MBT testing approach. The KAOS documents are also reflecting the test requirements needed for the test engineer, mostly by the CCB. Further, the test oracle is given and directly transformed into C#. However, the test oracle needs to be completed by the information whether or not there is a valid reconfiguration at hand. Developing an algorithm solving that problem is still a challenging task for the test engineer.

The CCB and the other artifacts from the first phase enable testing of SO mechanisms, as well as the CEI architecture. *R3* questions whether the CEI is testable. The implementation of the production cell and the SO mechanism follow the CEI architecture and, indeed, enable observability and controllability. The task of building the test scaffold was made easy by the clearly defined interfaces on test engineer and development engineer side. The different parts of the SO mechanism are identified and also clearly differentiated in their responsibilities. The model also enables isolation and disassembling. The disassembling is given by the defined parts of the SO mechanism, and the isolation is available by the model that can stub the not integrated parts. We used that concept successfully by stubbing the distribution phase of the SO mechanism and integrating it later on. Technically, the switch between stubbing and using the test driver is just one configuration parameter, if the functionality is implemented. Thus, answering *R4* and

R5, isolation, disassembling, and integrating is possible and applicable for testing SO mechanisms.

The test architecture embeds all the different concepts discussed and is helpful for the concrete implementation of the test approach. This test architecture has been successfully used as a foundation and skeleton for the test model in the production cell. It helps to organize and structure the model and its implementation. Thus, it is indeed enabling testing of SO mechanisms (*R6*).

Having the approach for MBT complete it was possible to start testing by randomly generating test cases, as shown above. The test cases revealed failures, and it was possible to execute the testing in full automation. The closed-loop enabled to use the information of the system state and map the actual state of the SuT. This information was primarily used for evaluation of a test case execution, but will also be used, as shown in the next chapter, for test case selection. Thus, to answer *R8*, the closed-loop MBT was able to reveal failures in the SO mechanisms. Further, the failures revealed were showing the need for BtB testing in this setting, as discussed above. The kinds of failures are traced back to errors, which are caused by a misunderstanding of the requirements. Thus, *R9* is answered with a clear yes.

6.6.2 Energy Grid—Testing a Disassembled, Continuous Self-Organization Mechanism

The energy gird case study is introduced in Chapter 2. The implementation and concepts of SO are the results of a six years DFG funded research program, resulting in three dissertations [6, 147, 153]. We classified the system and its SO mechanisms as continuous in Chapter 2.

Within this section, we address the research questions *R1-R7* of this chapter. For the evaluation, we use two different SO algorithms that will be tested in isolation, using the concepts of decomposition and isolation applied to a system with highly interwoven SO mechanisms. The requirements for this system, formulated in [6, 147, 153], are used to build the test system. As the SO algorithms used are already implemented and integrated into the system, we start with a thorough description of the two SO algorithms under test [52]. Afterward, the test system is created and described, and we conclude with the results.

Tested Self-Organization Algorithms

In this section, we present two SO algorithms, a decentralized approach (called SPADA) and a metaheuristic (called PSOPP). The aim of both algorithms is to partition a set of agents $\mathcal{A} = \{a_1, \ldots, a_n\}$ into pairwise disjoint subsets, i.e., partitions, that together constitute a *partitioning* at as minimal costs as possible. Concerning our case study, each AVPP represents a partition, and the set of all AVPPs corresponds to a partitioning. Because both algorithms make use of randomized decisions to find high-quality solutions in large search spaces, a testing approach has to deal with vast state spaces.

A Decentralized Algorithm for Partitioning Multi-Agent Systems SPADA [8], the ***S**et **P**artitioning **A**lgorithm for **D**istributed **A**gents*, solves the *complete set partitioning problem* (CSPP) in a general, decentralized manner. In the CSPP, the goal is to partition a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ into pairwise disjoint subsets, i.e., partitions, that exhibit application-specific properties. Because SPADA allows the definition of application-specific metrics, it can be applied to a variety of problems. In case a metric defines how well agents can work together on a common task, the CSPP is equivalent to coalition structure generation [128]. Since SPADA has been designed to solve the CSPP in general, it can be applied to these specific problems as well. This distinguishes SPADA from other centralized and decentralized approaches, which are often specialized to a specific problem in a specific domain. In the following, we give a summary of SPADA's basic functionality and characteristics. A more detailed description can be found in [8].

In SPADA, the agents use an internal graph-based representation of the current partitioning, called *acquaintances graph*, to solve the CSPP. All operations the agents apply to establish a suitable partitioning can, therefore, be mapped to graph operations. The nodes of the acquaintances' graph are the agents participating in the reorganization. Directed edges represent acquaintance relationships between agents. Together the acquaintances form an overlay network that restricts communication to acquainted agents, thereby lowering complexity in large systems. To indicate that an agent is not only acquainted with another but also in the same partition, edges can be marked. Partitions are thus defined by the transitive-reflexive closure of the binary relation given by the marked edges. Each partition has a designated leader that is responsible for optimizing its composition according to application-specific criteria. An example of such an acquaintances graph is depicted in Figure 6.7 (more details concerning the acquaintances' graph can be found in [8]).

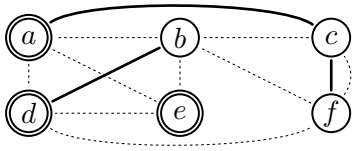


Figure 6.7. An exemplary acquaintances graph for a system consisting of six agents (cf. [8]): Agents are represented as nodes and acquaintances as directed edges, e.g., d is acquainted with b and f. Marked edges (symbolized as solid arcs) indicate that their tail and head belong to the same partition. In this example, there are three partitions $\{a, c, f\}, \{b, d\}, \{e\}$ with leaders a, d, e.

Each leader periodically evaluates if it is beneficial to integrate new agents or to exclude some of its members (e.g., concerning our case-study, to improve the equal distribution of unreliable power plants among AVPPs), to improve its partition. The latter can be beneficial in case of reorganizations that require to create new partitions, e.g., if a partition's or an agent's properties have changed so that the partition's formation criteria no longer favor including the agent. The integration and exclusion of agents are implemented by modifying the edges in the acquaintances' graph.

To decide about termination, leaders periodically evaluate application-specific termination criteria. These are formulated as constraints which can also be monitored at run time to trigger reorganization. If the termination criteria are met, the leader marks its

partition as terminated. As long as a partition is marked as terminated, its leader does not change its structure. However, the termination labeling is removed if the partition is changed from outside, i.e., if one of its members is integrated into another partition. This characteristic allows SPADA to make selective changes to an existing partitioning, which is very useful in dynamic environments. It has been shown empirically that SPADA's local decisions lead to a partitioning whose quality is within 10% of the optimum [8].

Concerning our case study, each leader instantiates a new AVPP agent as soon as all partitions terminated. The AVPP then assumes control of all power plants in the partition. In case of a reorganization, the acquaintances' graph is created from the existing system structure.

A Particle Swarm Optimizer for Partitioning Multi-Agent Systems *PSOPP* [10], the ***P**article **S**warm **O**ptimizer for the **P**artitioning **P**roblem*, is based on *Particle Swarm Optimization* (PSO) [86], a bio-inspired computational method and metaheuristic for optimization in large search spaces. In PSO, a number of particles concurrently explore the search space in search of better candidate solutions by modifying their current positions (at random or by approaching other candidate solutions) as long as a specific termination criterion is not met. During this process, each particle's current position represents a specific candidate solution. To be able to improve the quality of candidate solutions in a target-oriented manner, each particle Π_i is aware of its best-found solution $\mathcal{B}_i$ and the best-found solution $\mathcal{B}_{\mathcal{N}_i}$ in its neighborhood $\mathcal{N}_i$. The algorithm's outcome is the global best-found solution $\mathcal{B}$.

PSOPP solves a variant of the CSPP in the presence of *partitioning constraints* that constrain feasible partitions concerning a minimum s_{min} and a maximum s_{max} size as well as a minimum n_{min} and a maximum n_{max} number of partitions. Therefore, the test oracle additionally has to check—without interfering with the algorithm, by evaluating the logged data of the algorithm and not locking it during execution—if the interim results and the resulting system structure satisfy the partitioning constraints. In PSOPP, each particle represents a partitioning that satisfies the partitioning constraints. The central idea is to allow the particles to move around the search space by using the basic set operations *join*, *split*, and *exchange* to come to a solution. The join operation creates the union of two partitions. The split operation divides an existing partition into two non-empty subsets and the exchange operation exchanges elements between two partitions. Particles can apply these operations at random as well as in a target-oriented manner. The primary purpose of the former case is to enable the particles to explore the search space by randomly modifying their represented partitioning, i.e., their position. The latter case, in contrast, allows particles to exploit existing candidate solutions by approaching other candidate solutions in promising regions of the search space. Since PSOPP's operations are defined in a way that their application always maintains solution correctness, it combs through a search space that only contains correct solutions. This is advantageous concerning its performance.

Similar to SPADA, PSOPP can be customized to a specific application by devising an appropriate fitness function that assesses the quality of solutions and thus steers the search for them. Due to these characteristics, PSOPP can be applied to many different

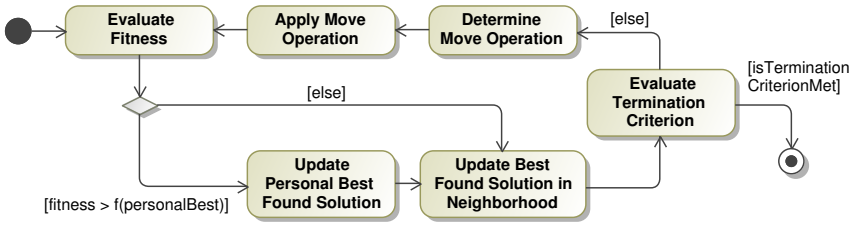


Figure 6.8. Actions performed by particles in each iteration, taken from Anders et al. [10].

applications in which solving the partitioning problem considered in this thesis is relevant and global knowledge is available.

Having specified valid partitionings by means of $n_{min}, n_{max}, s_{min}, s_{max}$ as well as the particles' attitude towards exploration and exploitation by fixing some parameters that influence the probability that particles make a random move or approach other candidate solutions, PSOPP creates a predefined number of particles at random or predetermined positions in the search space (the set of particles does not change at run time). The latter is suitable when a reorganization of an existing system structure has to take place: If the current structure does not contradict the partitioning constraints, it can be used as a starting point for the self-organization process. Mixing predefined and randomly generated initial partitionings allows to hold up diversity. When searching for an initial system structure, particles are created at random positions. As long as a specific termination criterion is not met, a particle Π_i performs the following actions in each iteration; these are also depicted in Figure 6.8:

1. Evaluate the fitness $f(\mathcal{P})$ of the represented partitioning $\mathcal{P}$.
2. If the particle's fitness $f(\mathcal{P})$ is higher than the fitness $f(\mathcal{B}_i)$ of its best found solution $\mathcal{B}_i$, set $\mathcal{B}_i$ to $\mathcal{P}$. Further, inform all other particles Π_j that contain Π_i in their neighborhood $\mathcal{N}_j$ about the improvement so that they can update $\mathcal{B}_{\mathcal{N}_j}$, i.e., the best found solution in their neighborhood.
3. Update the best found solution $\mathcal{B}_{\mathcal{N}_i}$ in Π_i's neighborhood $\mathcal{N}_i$.
4. Stop if the termination criterion is met.
5. Otherwise, randomly opt for the direction in which to move, i.e., choose whether a random move or an approach operation should be applied. In case of an approach operation, also determine the position (i.e., $\mathcal{B}_i$ or $\mathcal{B}_{\mathcal{N}_i}$) that should be approached.
6. Determine the new position $\mathcal{P}'$ by applying the selected move operation to $\mathcal{P}$.

Once all particles terminated, PSOPP returns the best found solution $\mathcal{B}$. Possible termination criteria are, e.g., a predefined amount of time, a predefined number of iterations (i.e., moves through the search space), a predefined threshold for the minimum fitness value, or a combination of these criteria.

Building the KAOS, System, and Environment Model

The starting point of the development is the main system's goal of the SuT. The primary system goal of the energy grid is to maintain a stable grid. This goal is refined during

the analysis of the requirements to system goals, like the ones shown in Figure 6.10. Indeed, this is only one excerpt of the overall goal model for the energy grid but is of high concern for the SO algorithms described before that are tested in isolation in this evaluation. The described requirements in Figure 6.10 are concerning the scheduling and the adjustments to be made by the power plants and the AVPPs in the system for supplying the energy demanded. The obstacles (marked as red rhombuses) are challenging these goals and demand for self-organization. The self-organization is performed by grouping the AVPPs in order to fulfill these properties. Accordingly, the OCL constraints in the requirements incorporate this fact. One of the constraints is for example capturing the maximum scheduling time as follows:

context AVPP **maxspan(3,8)** schedulingBelowThreshold:
self.schedulingTime < self.maximumSchedulingTime

The **maxspan** stereotype is an addition to standard OCL, that has been added for modeling this case study. It takes two arguments as input allowing to define the maximum number of violations in a timespan. This extension is incorporated into the test oracle synthesis presented in Chapter 3. The **maxspan** stereotype is therefore defined formally as follows:

$$s_i, s_{i-1} \in S; s_i \rightarrow s_{i-1} : |\phi(s_i)|_t^n = \begin{cases} true, & \text{if } n = 0, \\ false, & \text{if } n \neq 0 \wedge t = 0, \\ (\phi(s_i) \wedge |\phi(s_{i-1})|_{t-1}^{n-1}) & \\ \wedge(\neg\phi(s_i) \wedge |\phi(s_{i-1})|_{t-1}^{n}), & \text{otherwise} \end{cases}$$

where ϕ is the inner constraint that is evaluated at state s_i as the **inv** stereotype in OCL and n and t are the **maxspan** parameters. In each time step $\phi(s_{\text{current}})$ is evaluated and the result is saved. The implementation translated to in the test oracle uses a ring buffer with a fixed size of n for managing the historical values for evaluation. The introduction of this parameter is due to the history dependent on the behavior of the SO mechanism that is to be specified.

The completed CCB is transformed into a test oracle written in Java. However, as in the production cell case study before the oracle has to be completed by particular concerns of the SO mechanism. The additional checks are described later in this section.

Indeed, the schedulingBelowThreshold constraint is not the only constraint forming the CCB, the one shown is a representative element of all constraints, that has been built during the evaluation. The context of the constraints is defined in the domain model that is shown in Figure 6.9. That model is further used to build the system and environment model for the MBT setting.

Scaffolding and Implementation of the Test Model

In this evaluation, we used Java for describing the test model. That is possible since we do not need the extension of S# here for the environment faults. Describing the system, environment, and (in this case of continuous SO mechanisms) the test model is possible in different object-oriented languages, as claimed in this chapter. However, we

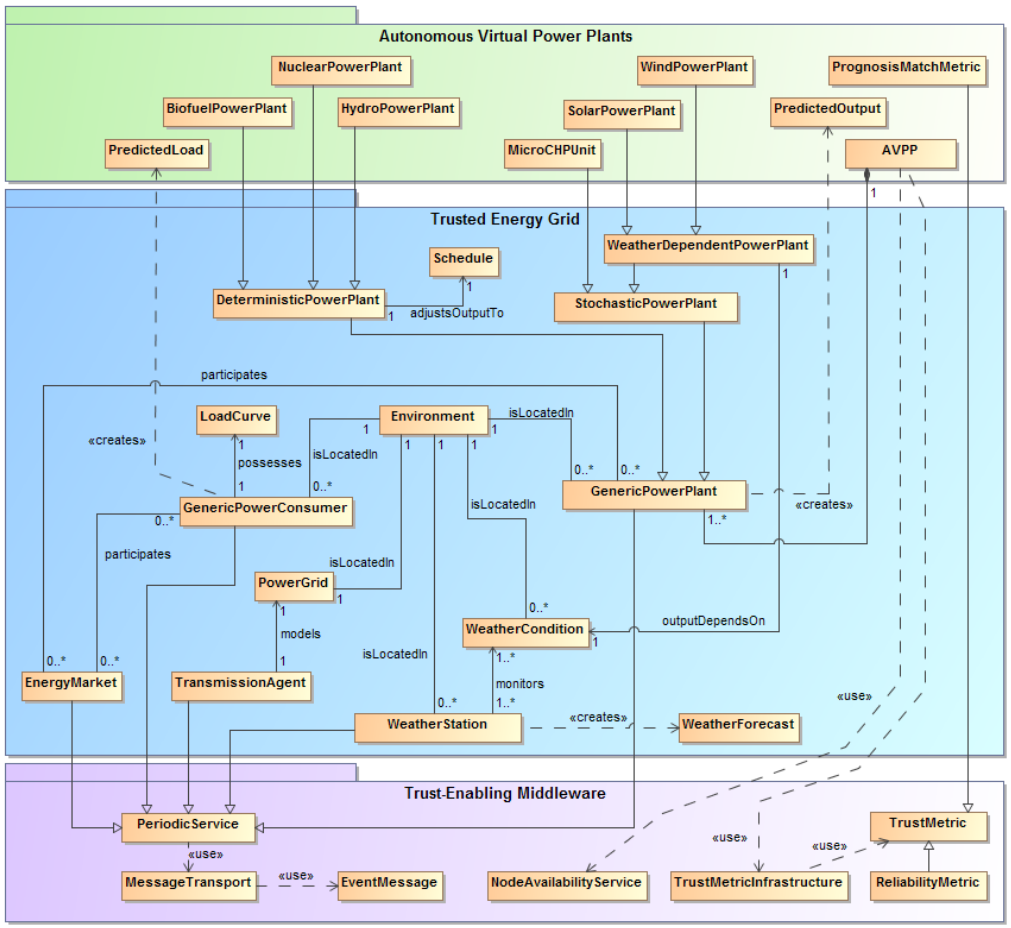


Figure 6.9. Showing a graphical representation of the system and environment model of the energy grid case study as an UML class diagram, taken from Anders et al. [9]. The model is divided into three package, for the sake of readability: the autonomous virtual power plants, the trusted energy grid, and the trust-enabling middleware. The focus on trust is due to the conceptual focus of the development of the system on using trust concepts for SO. However, despite this special case, the principles for designing a system and environment model are still valid.

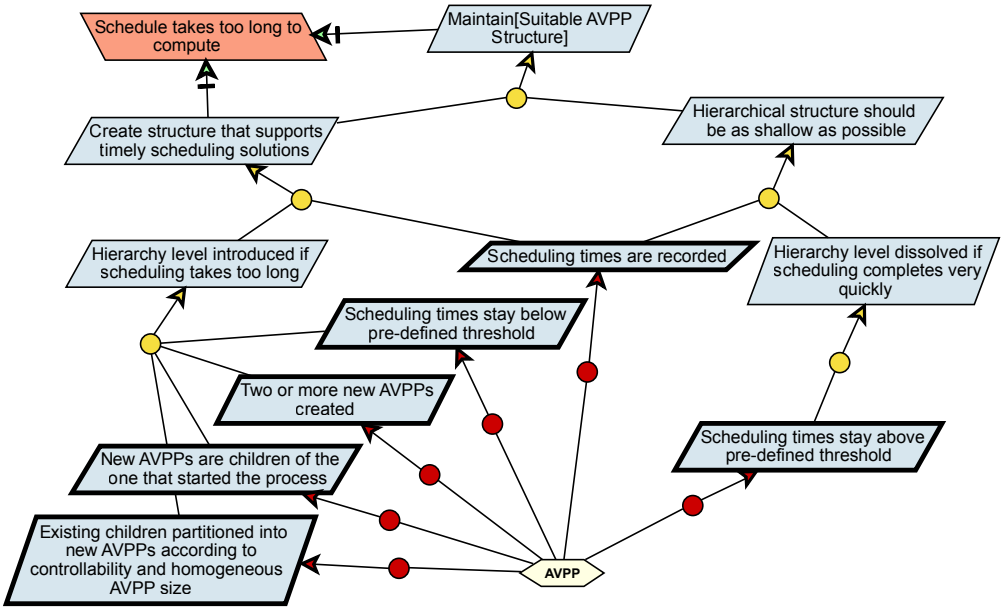


Figure 6.10. The shown excerpt of the KAOS model that has been designed for testing the energy grid's SO mechanisms. The agents are classes of the model in Figure 6.9. They are responsible for different requirements, in this case the AVPP is responsible for maintaining an organization structure to enable scheduling of the power plants. The requirements, which are not shown here (cf. Chapter 2 in Figure 3.3 for some concerning requirements) are challenging due to long lasting scheduling, leading to a need for SO. The same applies for the other requirements. For that reason, the corridor is defined (in parts) by the conjunction of the OCL constraints in the requirements that are specified here.

need an execution environment that can execute the model. Further, a mechanism for updating the model by reflecting the state is required in order to form a run time model. For testing the energy grid the same execution environment as for the SuT is used, it is called TEMAS [9]. We opted for TEMAS because it supports a stepwise execution out of the box, which allows the presented approach to monitor consistent states of the system at specific points in time. The mapping between the system state and the model state is done by calling the information from the SuT in the model at every step. The principle is the same as described for the S# model.

As foundation of the *system and environment model*, we used the domain model of the AVPP application, shown in Figure 6.9.

The *test model* describes the effect of different environmental conditions, such as weather conditions, on the power plants. Due to this dependency of weather conditions the power plants' predictions of their future output are inaccurate. The inaccuracy is affecting the suitableness of a power plant to an AVPP. This effect depends on the concrete type of agent, i.e., power plant. For this reason, we regarded four different agent types:

1. Solar panels
2. Wind turbines
3. Biogas power plants
4. Hydropower plants

Based on the assumption that the effect of changing environmental conditions, such as the global radiation, the wind speed, the available amount of biogas, or the water flow, is characteristic of a specific type of power plant, we generated different sets of agent groups. Further, the power plants' geographic location was taken into account. Considering an AVPP's prediction accuracy as a property resulting from the average prediction accuracy of its members, the system's goal was to maintain a structure of AVPPs that feature a similar prediction accuracy (cf. Chapter 2). As soon as the dissimilarity of the AVPPs' prediction accuracy exceeded a certain threshold (as a result of environmental changes), the power plants triggered a reorganization that should reestablish the similarity. We parametrized the model as follows:

- number of agents $\#ag$: between 2 and 1000
- agent group size: between 2 and $\#ag$
- number of agent groups: between 1 and $\left\lfloor \frac{1}{2} \cdot \#ag \right\rfloor$
- partition size: $2 \leq s_{min} \leq s_{max} \leq \#ag$
- number of partitions: $1 \leq n_{min} \leq n_{max} \leq \left\lfloor \frac{1}{2} \cdot \#ag \right\rfloor$
- 10 test sequences per test suite
- number of test cases per test sequence: between 50 and 1000
- number of states per EP: between 3 and 25

As SOuTs, we integrated the Java-based implementation of the partitioning algorithms SPADA and PSOPP. Both implement the IController interface of the test framework, presented in Chapter 5, that is used by the SO Algorithm Adapter to initiate the SOuT, request results, and ask the SOuT to adopt the new system structure (cf. Figure 5.2

in Chapter 5). The latter was implemented by sequentially moving the power plants contained in the calculated partitioning from their current AVPP into the corresponding new AVPP. Concerning the system structure, this procedure assures that every power plant is always contained in precisely one AVPP. If the last power plant was removed from an AVPP, this AVPP was dissolved. The description of the algorithms provided by Anders et al. [8, 10] as well as their implementation served as the *system model*. As explained in Section 6.3, we used this information to identify relevant parameters and suitable valid ranges for their parametrization. For SPADA and PSOPP, we identified the following parameters:

- SPADA
  - number of acquaintances per agent: between 1 and 20
  - number of agents each leader evaluates for integration into its partition: between 1 and 10
  - maximum number of agents a leader can integrate into its partition within a single step: between 1 and 10
- PSOPP
  - number of particles $\#P$: between 1 and 4
  - number of particles starting at the current partitioning: between 0 and $\#P$
  - probabilities $c_{rdm}, c_{\mathcal{B}_i}, c_{\mathcal{B}} \in [0, 1]$ (with $c_{rdm} + c_{\mathcal{B}_i} + c_{\mathcal{B}} = 1$) to apply a random move operator, approach the particle's best found solution $\mathcal{B}_i$, and approach the global best found solution $\mathcal{B}$, respectively
  - max. run time in seconds: between 1 and 10

After each reorganization, the test oracle checks if the algorithm's result complies with the definition of partitionings (i.e., each power plant must be a member of exactly one AVPP). These checks are performed once the algorithm indicates its termination as well as after the result has been adopted on the model. Only in case of PSOPP, the test oracle additionally evaluated the satisfaction of the partitioning constraints since SPADA does not allow to restrict valid partitionings concerning the size and number of partitions.

Fault Injection for Mutation Analysis of the Test Approach To evaluate the approach proposed in this thesis, we injected four faults into the SPADA and five faults into the PSOPP implementation. Our injected faults have in common that they do not cause the algorithm to throw an exception that has to be caught by the test oracle (i.e., smoke tests), but that their application can result in an invalid reorganization result or invalid system structure. In a preliminary evaluation that ran for about one week, we tested the SPADA and PSOPP implementation without injecting any faults. We did not observe any failures in the course of these tests. So we can be confident that the failures the test oracle reported during our subsequent evaluation can be attributed to a specific injected fault.

SPADA: Injected Faults. The first two types of SPADA faults (cf. SPADA-F1 and SPADA-F2) manifest in an incorrect transformation of the current system structure into SPADA's internal model of a partitioning, i.e., the acquaintances' graph (cf. Section 6.6.2). This false mapping results in an invalid reorganization result.

SPADA-F1/SPADA-F2

- *Description*: When creating the acquaintances' graph for a new reorganization from the current system structure, an arbitrary AVPP is not represented in the acquaintances' graph if the number of AVPPs is above (in case of SPADA-F1) or below (in case of SPADA-F2) a certain threshold. We set these thresholds to 100 for SPADA-F1 and to 5 for SPADA-F2.
- *Effect*: The resulting partitioning does not contain the power plants that have been members of the "forgotten" AVPP.

The two other types of faults we integrated into SPADA concern a functionality that is used to transform the result, given in the form of an acquaintances graph, into a set of sets. This functionality is used to provide the result to the SO Algorithm Adapter and as a preprocessing step to create the new AVPP structure.

SPADA-F3

- *Description*: In case the size of a partition exceeds a predefined threshold, arbitrary power plants are deleted from this partition until its size equals this threshold. In our evaluation, we set this threshold to 100.
- *Effect*: Some power plants are not represented in the partitioning.

SPADA-F4

- *Description*: In case the size of a partition exceeds a predefined threshold, this partition is replaced by a partition that is randomly selected from the partitioning. In our evaluation, we set this threshold to 100.
- *Effect*: Some power plants are not represented in the partitioning, whereas others occur two or more times.

All SPADA faults can be detected before the underlying system structure is changed. Given the way the result is transformed into a new system structure (it is ensured that every power plant is always a member of exactly one AVPP), these faults *cannot* be detected by only considering the distributed solution of the SO algorithm. For each type of injected fault, we are confronted with the problem of error masking.

PSOPP: Injected Faults. Regarding PSOPP, we modified the implementation of the move operations "random split" (PSOPP-F1), "random join" (PSOPP-F2), "approach split" (PSOPP-F3), "approach join" (PSOPP-F4), and "approach exchange" (PSOPP-F5) as described in the following listing.

PSOPP-F1

- *Description*: If a partition K is randomly split into two partitions L and M, an arbitrary power plant of L is replaced by another arbitrary power plant of M. This fault does

only occur if the size of L and M is below a threshold of t_1 or above a threshold of t_2.

- *Effect*: Concerning the resulting partitioning, a specific power plant is missing, and another occurs twice.

PSOPP-F2

- *Description:* If two partitions K and L are merged into a new partition M when applying the random join operator, either K or L is not removed from the partitioning. This fault does only occur if the size of K and L is below t_1 or above t_2.
- *Effect:* In the resulting partitioning, the power plants of either K or L occur twice as they are also contained in M.

PSOPP-F3

- *Description:* If a partition K is split into two partitions L and M, the resulting partitioning does not contain either L or M. This fault does only occur if the size of L and M is below t_1 or above t_2.
- *Effect:* In the resulting partitioning, the power plants of either partition L or M are missing.

PSOPP-F4

- *Description*: If two partitions K and L are merged into a new partition M when applying the approach to the join operator, one element is removed from M. This fault does only occur if the size of K and L is below t_1 or above t_2.
- *Effect*: In the resulting partitioning, a single power plant is missing.

PSOPP-F5

- *Description*: If some power plants are exchanged between two partitions K and L, one power plant of either K or L occurs in both resulting partitions M and N. This fault does only occur if the size of M and N is below t_1 or above t_2.
- *Effect*: In the resulting partitioning, one power plants occurs twice.

In our experiments, we used $t_1 = 2$ and $t_2 = 100$ so that the failures do only occur in certain situations. Note that the application of an injected fault does not necessarily yield an invalid result because an invalid candidate solution must be rated better than all other (possibly valid) candidate solutions found by the particles. This characteristic together with PSOPP's non-deterministic behavior exacerbates the detection of an injected fault.

Note that not all wrong results manifest themselves in an invalid system structure. Consider the following example illustrating error masking: Assume that the power plants a and b are currently members of the same AVPP. If a reorganization causes an invalid result that does not contain these two power plants, the test oracle detects a failure investigating the preliminary result. However, the resulting system structure is *valid* in case the minimum size of an AVPP is ≤ 2 and the maximum number of AVPPs is

	PSOPP						SPADA			
Injected Fault	F1	F2	F3	F4	F5	F5d	F1	F2	F3	F4
#EP States	11.68 (5.39)	15.77 (6.49)	12.30 (5.96)	13.74 (5.75)	15.41 (6.16)	14.25 (6.05)	16.42 (5.71)	14.72 (6.42)	16.62 (6.19)	15.44 (6.31)
#EP Transitions	127.42 (99.20)	222.06 (148.59)	143.31 (124.31)	169.78 (128.68)	210.22 (142.27)	183.28 (130.00)	230.83 (134.75)	197.09 (134.29)	240.56 (141.24)	212.72 (136.51)
%EP State Coverage	99.51 (2.33)	99.57 (2.08)	99.45 (2.50)	99.42 (2.71)	99.47 (2.35)	99.54 (2.31)	99.66 (1.77)	99.59 (2.15)	99.54 (2.35)	99.60 (1.98)
%EP Transition Coverage	52.31 (15.80)	45.37 (12.63)	50.57 (14.32)	47.53 (11.97)	44.76 (11.47)	47.57 (14.11)	43.98 (11.34)	47.46 (15.11)	43.52 (11.60)	45.34 (13.23)

Table 6.2. Statistical data concerning the number of EP states, EP transitions, as well as the coverage of EP states and EP transitions. All values are averages over the 700 generated test sequences per injected fault type. Values in parentheses denote standard deviations.

not exceeded. This is because a and b remain in their old AVPP if they are not contained in the provided result.

Discussion of the Test Results To be able to make a clear statement which types of faults can be found by the presented approach, only one specific type was injected during the execution of a single test sequence. All in all, we injected 10 different types of faults: SPADA-F1 to SPADA-F4, PSOPP-F1 to PSOPP-F5, and an additional variant of PSOPP-F5, called PSOPP-F5d, which we will explain in more detail in the course of this section. For each type, we generated 70 test suites, each containing 10 test sequences, resulting in 700 test sequences per fault type and a total number of executed test sequences of 7000. Overall, we generated 3,679,326 test cases, corresponding to an average of 367,932.60 per fault type. As shown in Table 6.2, this high number of test cases allowed us to obtain an EP state coverage of more than 99% and an EP transition coverage ranging between approximately 44% and 52% on average for all fault types. In the course of the execution of a single test sequence, the presented approach could register multiple failures.

For this evaluation, we distinguish how the test oracle revealed a failure. In Figure 5.2 in Chapter 5, this is depicted by two different interfaces: one working on the agent, i.e., the distributed results, and one on the SO algorithm, i.e., the computed solution, as well as interim results. The first one is the here called black-box view and the latter one the gray-box view of the oracle. Distinguishing these two views will show the necessity in this case study to check the interim results for revealing as early as possible a fault. The gray-box view necessitates extending the automatically generated test oracle by manual checks based on the algorithmic approach of the SO algorithm. As Table 6.3 shows, the presented approach was able to detect every kind of injected fault and our evaluation results support our claims made concerning the need for a gray-box view: (1) All failures detected using the black-box view are also detected using the gray-box view, and (2) the SPADA faults cannot be detected using the black-box view. The fact that not all PSOPP faults that were disclosed using the gray-box view were also registered using the black-box view (between 0.00% and 34.44% on average) also demonstrates the problem of error masking in SOAS. Except for PSOPP-F4, the percentage of detected failures using the gray-box view (between 50.00% and 82.31% on average) outmatches

	PSOPP						SPADA			
Injected Fault	F1	F2	F3	F4	F5	F5d	F1	F2	F3	F4
#Agents	523.53 (310.53)	529.66 (292.23)	473.29 (269.35)	511.83 (287.62)	491.26 (300.28)	477.64 (294.48)	520.29 (286.29)	509.63 (289.01)	494.90 (297.69)	447.40 (279.09)
#Agent groups	39.80 (67.18)	37.42 (48.83)	20.09 (27.72)	36.05 (64.05)	30.2 (47.79)	32.74 (48.77)	31.86 (57.70)	35.12 (47.42)	31.30 (58.27)	38.85 (45.35)
%Test sequences without failure	79.97 (40.05)	84.55 (36.17)	91.42 (28.03)	96.28 (18.94)	97.86 (14.49)	70.22 (45.76)	97.13 (16.69)	84.22 (36.48)	85.69 (35.04)	90.13 (29.85)
#Test cases per test sequence	520.01 (281.50)	523.71 (275.73)	531.53 (273.61)	521.19 (278.14)	517.23 (272.25)	518.54 (270.26)	548.38 (278.18)	535.62 (271.43)	511.46 (277.29)	518.51 (282.27)
%Applied test cases per test sequence	81.37 (38.39)	85.69 (34.44)	92.74 (25.62)	96.77 (17.11)	98.62 (11.16)	73.90 (42.75)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)
#Reorganizations per test sequence	73.29 (64.40)	77.68 (62.08)	91.87 (59.90)	91.87 (60.24)	90.01 (56.36)	53.35 (61.19)	100.00 (58.20)	87.64 (62.46)	83.60 (60.35)	87.75 (61.30)
#Failures per test sequence	0.36 (1.89)	0.19 (0.53)	0.13 (0.52)	2.29 (16.03)	0.02 (0.15)	0.54 (1.03)	0.03 (0.17)	0.16 (0.36)	83.49 (60.39)	0.12 (0.35)
#Failures	252	130	91	1603	16	376	20	110	58443	84
%Undetected failures	44.44	17.69	34.07	97.75	50.00	40.96	0.00	0.00	99.84	9.52
%Detected failures (gray box)	55.56	82.31	65.93	2.25	50.00	59.04	100.00	100.00	0.16	90.48
%Detected failures (black box)	53.17	80.00	57.14	2.25	31.25	51.33	0.00	0.00	0.00	0.00
%Test sequences with failures detected in gray box only	4.29 (20.33)	3.70 (18.97)	13.33 (34.28)	0.00 (0.00)	34.44 (45.63)	9.74 (27.63)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)	100.00 (0.00)
%Test sequences with failures detected in black box only	0.00 (0.00)	0.93 (9.62)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)	0.00 (0.00)
Depth of first detected failure	8.56 (22.82)	17.50 (62.33)	7.55 (9.12)	44.04 (109.78)	30.87 (74.51)	24.28 (75.26)	3.85 (0.49)	4.66 (1.64)	9.96 (14.38)	13.20 (16.06)
Depth of first detected failure (gray box)	8.56 (22.82)	17.63 (62.61)	7.55 (9.12)	44.04 (109.78)	30.87 (74.51)	24.28 (75.26)	3.85 (0.49)	4.66 (1.64)	9.96 (14.38)	13.20 (16.06)
Depth of first detected failure (black box)	8.77 (23.30)	13.64 (46.08)	6.08 (5.12)	44.04 (109.78)	90.27 (173.32)	22.83 (71.90)	N/A	N/A	N/A	N/A

Table 6.3. Statistical data concerning the number of agents, the number of agent groups, the number of test cases, the number of reorganizations, as well as the occurrence, detection, and depth of failures. All undetected failures (see "%Undetected failures per test sequence") can be attributed to error masking. "#Failures" and "#Failures per test sequence" refer to the number of faulty intermediate states the corresponding SO algorithm entered (note that this information is provided by our fault injection mechanism and not by the oracle that can only check for the validity of final states, i.e., reorganization results). All values are averages over the 700 generated test sequences per injected fault type. Values in parentheses denote standard deviations.

the percentage of detected failures using the black-box view in all cases (between 31.25% and 80.00% on average). These observations indicate the need for gray-box interfaces in the context of testing SO algorithms.

The relatively high number of test sequences in which no failure was detected (between 70.22% and 97.86% on average) highlights the need for directed testing that can deal with the vast search space more efficiently. For PSOPP, the different numbers of applied test cases per test sequence reflect the difficulty of disclosing a specific type of fault using the black-box view. In the case of SPADA, all test cases were applied since the injected faults cannot be detected using the black-box view. Another indicator for the difficulty

of finding a specific fault type is the depth of the first detected failure (i.e., the index of the test case in which the first failure was detected). Here, we see significant differences among the different SPADA and PSOPP fault types. This is because the system not only has to be pushed into a faulty intermediate state, but the reorganization also has to end in a faulty state that can be detected by the oracle. This challenge becomes clearer when taking a look at the percentage of undetected failures due to error masking, which ranges from 17.69% to 97.75% for PSOPP and between 0.00% and 99.84% for SPADA. Together with the average number of reorganizations per test sequence—that, except for SPADA-F3, significantly outmatches the average number of failures per test sequence—these observations illustrate the difficulty of testing SOAS.

We observed that especially PSOPP-F5 had a relatively high number of executed test sequences without detected failures (97.86% compared to an average of 88.67% over all other types of fault). We, therefore, decided to use PSOPP-F5 to investigate the influence of directed testing on the presented approach's ability to disclose a fault. To study this effect, we introduced an additional fault type PSOPP-F5d that is equivalent to PSOPP-F5 but uses a specific system configuration: To increase the chance of applying PSOPP's approach exchange operator (an operation that is usually only applied in rare cases), we set the allowed number of partitions in PSOPP-F5d to $n_{min} = n_{max}$. Using this parametrization, PSOPP has no choice but to apply the random exchange or approach change operator, because the split/join operators increase/decrease the number of partitions by one. Note that this measure does not increase the code coverage (PSOPP also applied the approach exchange operator in case of PSOPP-F5), but the chance that the fault can be disclosed given the algorithm's non-deterministic behavior: In approximately the same number of executed test cases, PSOPP-F5d entered a faulty intermediate state about 23 times more often than PSOPP-F5. The benefit of directed testing is further reflected in an increase of the percentage of a faulty end state in case of a faulty intermediate state from 50.00% for PSOPP-F5 to 59.04% for PSOPP-F5d. Consequently, the number of applied test sequences without detected failures dropped from 97.86% to 70.22%.

Summarizing, although the presented approach was able to find every type of injected fault, the high number of needed test sequences for failure detection demonstrates that the automatic generation of test suites from EPs and influence functions is especially useful for testing SOAS. This is mainly due to the non-deterministic behavior of SO algorithms that are not directly addressed in the here executed test case generation procedure. However, our evaluation results also showed that a combination of model-based and random generation techniques is effectively detecting specific types of fault. In particular, the gray-box interface is an important feature to mitigate the effect of error masking. For a more efficient generation of the test cases, concepts are presented in Chapter 7.

Discussion of the Research Questions *R1-R4* and *R5-R7* We will now discuss the research questions *R1-R7*, that are answered by having established the described evaluation. With confidence, *R1* is answered with yes, as we showed how the specification was enabled by the concepts of the CCB and the KAOS methodology. The results are

not only the definition of the responsibility of the investigated SO algorithms but also a test oracle for the requirements for testing. Next, the outcome of a systematic analysis of the system's requirements leads also to a system and environment model that is completed by a test model formulated with EPs. This is forming a solid foundation for the overall test model and answers *R3* positively. *R4* is questioning the isolation. This is of particular interest in this evaluation since we only focused on testing the SO algorithms and the mechanisms are highly complex and interwoven with the system. Nevertheless, it was possible to establish the necessary scaffolding, as described in the approach presented in this thesis. We showed the test results and the results of mutation analysis, indicating the abilities of the approach to be executed in order to be able to reveal failures, having an affirmative answer for *R4, R6,* and *R7*.

6.6.3 Load-Balancing Web-Service—Evaluating the Test Approach in a Controlled Experiment

The Znn.com case study, described in Chapter 2, has been developed in a research project of the Carnegie Mellon University, resulting in the dissertation of Cheng [37]. It serves the evaluation research question *R13*, where we are going to investigate how useable the presented concepts are for test engineers that are not familiar with testing SO mechanisms. For this purpose, a controlled experiment was carried out: A student, familiar in the area of software testing, was challenged by first developing a simplified version of the application case of the Znn.com load-balancing web-service, as specified by Cheng et al. [37, 38]. The implementation was done in C# and the .NET environment. Then he was asked to test the developed SO mechanism in the Znn.com system, based on the concepts presented so far in the thesis, using the S# environment.

Next, we will elaborate and discuss the results of this experiment concerning the usability of the presented approach for MBT of SO mechanisms. The setting of the experiment was the following: The student was equipped with the documentation and specification of the Znn.com system written by Cheng et al. [37, 38]. Further, he received the documentation of the MBT approach for SO mechanisms as described in this chapter. Along with this description, he was granted access to the software repository of the production cell test implementation as well as a half-day tutorial on the testing approach. The student was asked to write a development diary where he documented his development steps and the experiences, troubles, and questions he had during the development. This development diary was evaluated and discussed twice a week during the development period of two months. The following results are the condensed experiences gained from this experiment.

Development and Implementation of the SO Mechanism

The setup of the implementation provides the following components in an *N-tier-client-server* architecture:

- Client: responsible for generating different kinds of requests to the system
- Server: responsible for processing a request of a client

- Proxy: responsible for scheduling the requests to the servers and also for organizing the servers' activity, i.e., the central point for the SO mechanism

For the implementation, the proxy and server components have been developed as a .NET application, where the servers and the proxy are running on the same machine. Both the client and the server are simple console applications, where the client can access the server application. Both run on the same computer and a simulation has been established, that can send requests and get the requests back with an updated state. The state of the request was responsible for tracking the process and the timely behavior. The following states have been introduced to the request for tracking:

- idle
- request processed by the proxy
- request processed by the server
- request with low fidelity completed
- request with medium fidelity completed
- request with high fidelity completed

The decision was first to keep the implementation of the fundamental structure as simple as possible to be able to focus on the SO mechanism. Thus, the processing of the requests at the server was only a fixed amount of time set at the server when a request is received. This amount of time can vary in order to simulate different kinds of servers. The student implemented a slow, medium, and fast server edition to be configured. Each server is set with a cost value that is correlating with the response time. The configuration of the system includes a number of different kinds of servers as well as the total number of servers. The requests at the client are generated with a fixed or a random frequency. The interval for the random frequency is also configurable. Further, the number of clients is due to a configuration.

The SO mechanism was developed according to the specification in Cheng [38] with slight adaptations. The environmental influence factors that we considered for the SO mechanism are the servers and there ability to perform. The servers might fail in total, the server might not be activated or deactivated, and the server might fail at obeying the configured response time. The SO mechanism is located in the proxy component and is based on a rule set consisting of the following rules:

- Detect failing servers and replace them adequately.
- Provide consistency of capabilities in the web-service by activating or deactivating the server according to a demanded fidelity (high, medium, low).
- Provide an optimal configuration according to the costs of a server in the network by activating or deactivating servers.

Having the configuration, the first experiments showed a similar behavior as described by Cheng [38]: The SO mechanism is smoothing the peaks of the costs and fidelity by adapting the structure of the system, compared to a solution with no SO mechanism.

Challenges in the Implementation The challenges for the student have been to work in the case study, the concept of SO mechanisms, and the implementation in .NET. In different iterations, the system was implemented, starting with conceptual planning and a step-wise refinement of the implementation. It was difficult to find the right abstractions from the actual web-service to an implementation that is focused on a few SO aspects. Overall, the development took one month, including the conceptual phase. The support for the student by the documentation of .NET and the specification of Cheng [38] was enough for a starting point. However, the implementation and conception of the SO mechanism required seasoned support.

Development and Implementation of the Test Framework

For the implementation of the test framework, the starting point was given by the approach of describing the requirements in the form of a KAOS model. These models have been sketched on flip charts and have been used for discussing the detailed requirements for the SOuT. This modeling approach turned out to be very helpful to foster a discussion about forming the CCB. The primary goal of the system is to provide news content. The following high-level goals have been derived from that goal:

- The proxy can measure the demand
- The proxy can activate or deactivate servers to serve the demand
- The server provides news content within their abilities
- The overall fidelity is medium

The introduction of obstacles enabled the student to identify different needs for adaptation and assign them to the proxy agent. The domain model that has been created was rather simple, consisting of the clients that are connected to a proxy that is connected to servers with the properties specified to enable the decisions for adaptation. That are mainly the availability, the fidelity and the state of the request by the client. The constraints in the requirements concern the following aspects:

- If a server fails it is adequately replaced
- If a server is no longer able to fulfill the demanded fidelity it is replaced, supported by another server, or stopped.
- If the cost is above a threshold no new server is activated.
- If a cheaper server (or server set) is available it replaces the current one.

The formulation in OCL constraint was not performed by the student since it was decided to formulate the test oracle directly. That was caused by the rather simple CCB. Thus, the insight gained was: the process of forming the requirements and generating the oracle might also be performed manually. That is also extremely helpful, as it is structuring the overall development process.

After having formed the CCB and having the first part of the test oracle, the check was implemented whether or not there is a valid configuration. In the given, simplified case study the algorithm is more straightforward than for instance in the production cell but was not straightforward to solve for the student at first. The solution was to check the

available resources and compute the demanded resources in the current situation by abstracting from the concrete server and just stating how many high, low, and medium servers are needed and compute the cheapest set. That algorithm runs on the model. The model is an implementation in S# classes for the three classes described before. The client implements a random generator for requests, as it serves as a test driver. The other classes are incorporating the web service and demanding information about the current state. Thus, it was necessary to enhance the implementation by getting an interface to retrieve the internal state and map it to the model. Further, we needed another interface for activating the environment faults on the servers, which are activated from the model. The needed surgery on the SuT's environment was highly invasive in order to execute the tests. The effort for test automation and enabling it was the most time-consuming part of the development of the test approach.

For specifying the test model, the student had to describe environment faults. That was for him somewhat unintuitive. However, the CCB guided him very well in this process. The entire environment faults are activated afterward randomly in a simulation started in S#.

The insight by the student was that the understanding of SO for the case study was much better from the testing perspective as from the development perspective, as it was more guided. Thus, he was able to reveal failures of misinterpretation from the implementation at this point.

Challenges in the Implementation Some of the challenges in the implementation of the test framework have already been mentioned. Formost, the automation and connection of the SuT's environment and of the SuT itself was technically demanding and time-consuming. The design of test cases as failures was counter-intuitively for the student, causing some initial problems. However, this problem was solved by first unfolding the the constraints of the CCB which support the task of defining test cases.

Discussion of the Research Question *R13*

The question *R13* challenges the usability of the approach presented in this chapter. The usability of the approach is the effectiveness, efficiency with which a test engineer that is new to testing SO mechanisms can test an SO mechanism. The answer to this question is supplied by the controlled experiment presented here. Indeed, having only one participant seems not statistically representative and demands for a more extensive experiment. However, it is still possible to gain insights on the usability aspects of the approach and about its strengths and weaknesses. The experiences made and documented are leading to the following observations: Establishing and implementing a full test automation, a test case generation, a test cases evaluation, and a test scaffolding is a challenging and time-consuming task. The observation made in this particular experiment and the creation of the test frameworks for the other case studies is that the effort is more or less the same as for the development of the tested SO mechanism. The systematic approach builds upon the KAOS model for deriving the requirements and the test oracle as well as deriving a first system and environment model. This process

turned out to be very helpful. Still, an in-depth understanding of the intention of the SOuT is needed. The creation of the KAOS model is well documented and supportive for carving out the intended behavior of the SOuT. However, the particular formulation of goals and constraints showed to be challenging and demands experience in developing SOAS. The model supported the necessary discussion and elaboration. Still, there is much room for interpretation of requirements and goals that are ultimately defined by the CCB. That strengthen the need for the BtB testing concept presented in this chapter. It has shown to be a valuable tool for aligning the requirements of the SuT.

The implementation of the test framework demands a skilled programmer, as establishing the scaffold is technically demanding. That effort was underestimated at first by the participant.

Describing the test model with environment faults turned out to be more challenging than expected for the student. That was mainly due to an inconvenient form of defining test cases. However, the conceptual connection with the CCB helped here to form the test suite.

Overall, it was possible to establish a fully automated and proper test framework for testing SO mechanisms within the experiment.

6.6.4 Pill Production—Investigating Reusability and Generalizability of the Test Model in Resource-Flow Systems

On the one hand side, we use for this evaluation the specification of the self-organizing pill production, as described in Chapter 2 that is an output of a research project of the University of Nottingham and published by Chaplin et al. [30]. On the other hand, we use the work by Seebach et al. [145] for the concept and design of resource-flow-oriented, self-organizing systems, the Organic Design Pattern (ODP). The latter one is implemented for the production cell, the case study that has already been tested and evaluated in detail above. The production cell is just one instance of the meta-model for resource-flow systems, as shown in Figure 2.4 in Chapter 2. The case of the pill production turned out to be another possible instance, which maps on that general meta-model. Figure 6.11 shows this mapping by an instantiation of the ODP meta-model for the pill production. The research question to be answered in this evaluation case is *R10*: how well does generalization apply in the MBT concept for different SuTs?

For this purpose, the complete test models for the pill production and the production cell have been developed twice. First, without any generalization, both models have been developed (by the same developer) complete and independently from the other models in S#. Second, the ODP meta-model has been implemented in S# and was used for the second implementation of the pill production and the production cell. The results, in source lines of codes (SLOC) are shown in Table 6.4. Overall, the effort for generating the ODP meta-model was quite significant, the development of the over 4k SLOC took over four weeks. However, the development effort, shown in SLOC, of the other models, derived from that meta-model are just around 60% the size as without generalization. Most of the reduction has been achieved in the system and environment model.

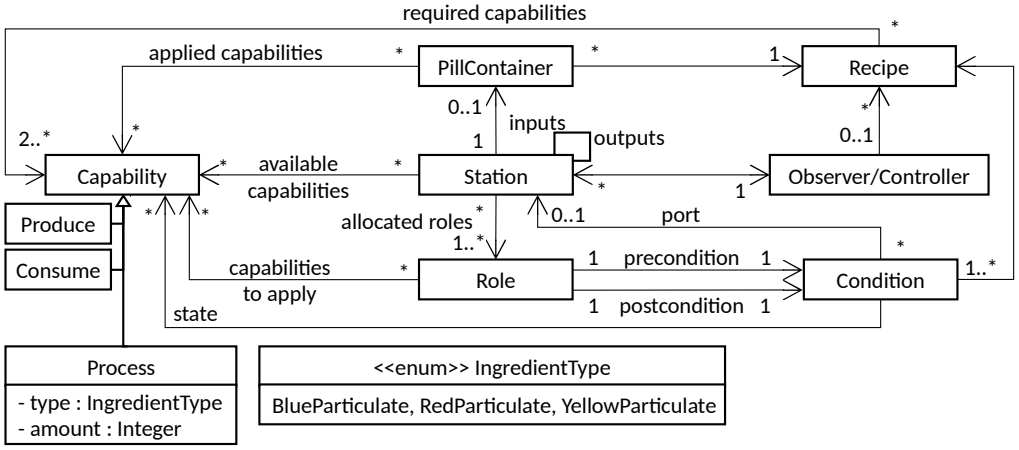


Figure 6.11. This UML class diagram shows the instantiation of the ODP for the pill production case study.

	ODP	Pill Production	Robot Cell
SLOC without Generalization		1,131	2,181
SLOC with Generalization	4,101	684	1,600

Table 6.4. The table shows the source lines of code (cleaned by removing blank lines and comments) of the models for the pill production and the production cell. The first line shows the numbers without generalization, i.e., the models are independently developed. The second line shows the numbers with the ODP meta-model and the reuse of its concepts in the other two models.

The abstraction ability of the MBT approach is, in this case, an enabler for generalization in testing. The effort for generating the models, but also for increasing the quality and the usability of the models by using an established meta-model is beneficial for testing SO mechanisms. That is shown by the fact that the size of the model using the ODP model was less than 60% of the ones used without meta-model. However, for providing more evidence on this assumption more empirical results are needed, that is due to future investigations. Nevertheless, the first result is auspicious and reveals high potential in building generalizable test models. One example that shows the potential is a new version of an SO mechanism that is working in a decentralized fashion, as described by Anders et al. [7]. The implementation of the SO mechanism can be solved generically for the resource-flow systems in the ODP. Testing the implementation is now possible for the production cell case study as well as for the production cell without additional effort, if it is integrated into the generic part of the model. Thus, it is possible to test one generic SO mechanism within two different environments without additional code for the model. The ability to introduce generalization comes with different merits: The meta-model can be established and form a reliable foundation for further developments of testing endeavors. The development effort is reduced and more guided by relying on the meta-model.

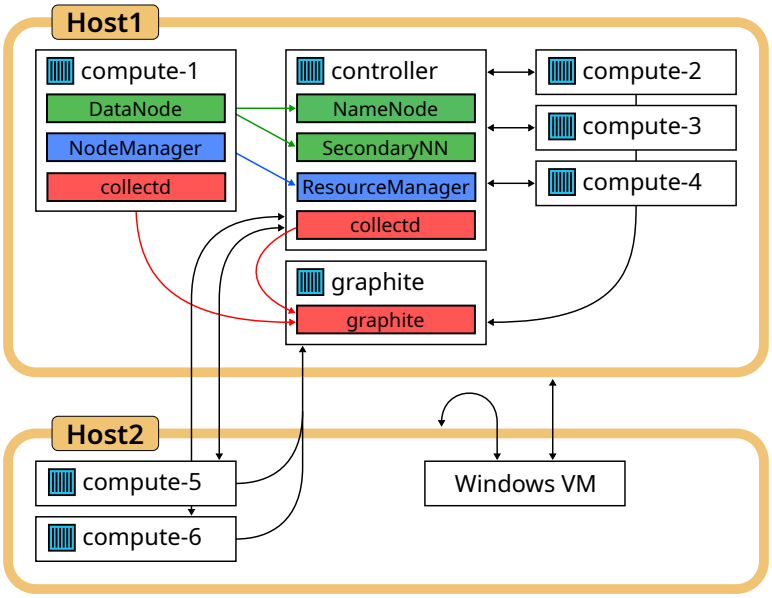


Figure 6.12. The deployment configuration of the Hadoop evaluation. The Hadoop application is distributed among two hosts that are organized in a docker-swarm. The SOuT is hosted in the controller, adapting the configuration of the ResourceManager to the current environmental conditions. The graphite component is deployed at each compute node for gathering information from the system. Further, the Windows VM is hosting the test framework.

6.6.5 Apache Hadoop—Testing an Industrial Case Study in Full Integration

Hadoop is one of the most popular and widely-used software platforms for big data processing and for applying the MapReduce paradigm to a large number of different applications and workloads. The performance of its application depends on the configuration of a bunch of parameters which need to be tuned for a specific task or workload. The *YARN* (*Yet Another Resource Negotiator*) resource manager is the component of Hadoop which is responsible for scheduling and controlling the workload within the cluster of compute nodes. The parameterization of *YARN* is decisive for the job performance. The best practice for setting the parameters is a best-effort configuration that is based on experience or static profiling, relying on apriori knowledge about the job. Zhang et al. [178] developed a self-adaptive component on top of *YARN*. It is an implementation of the *MAPE* architecture (cf. [87]), i.e., a control loop that *m*easures, *a*nalyzes, *p*lans, and *e*xecutes adaptation of the parameter setting of *YARN*. The authors showed that they can speed up the Hadoop instance up to 40% in a volatile environment compared to the best effort solution. We use the implementation of Zhang et al. [179], available at GitHub [150], which implements the concepts of Zhang et al. [178].

The implementation is deployed in a docker-swarm that uses two desktop computers equipped with Intel i5-4690 processors with 4 cores, 16 GB RAM, 512GB SSD, and Ubuntu 16.04 LTS as the operating system. The concrete deployment configuration can be seen in Figure 6.12.

For the sake of simplicity, we only use a subset of the overall requirements that are extracted from the Hadoop documentation [61] as well as the additional requirements of the adaptive extension documented in [150, 178, 179]. Since our test approach presented in this chapter is focused on functional testing, only the functional requirements of the *YARN* application are considered. The following functional requirements are used in the case study and are implemented in an CCB:

- A task will be completed, if it is not canceled
- No workload is allocated to inactive, defected, or disconnected nodes
- Parameters of the configuration are updated by the adaptation loop if a certain rule applies
- Defects or disconnections are recognized

The requirements have been translated into the CCB using the approach presented in Chapter 3.

S# Test Model for Hadoop Test Automation

A first step is to build the model in the modeling language S#. The model is used for the whole test process, i.e., the input generation, test execution, test evaluation, and the judgment. As the model is an executable run time model, it further incorporates the test driver.

KAOS Model First, it is necessary to form the KAOS requirements model along with the constraints in order to derive the test oracle. A key advantage of transforming these constraints to evaluation functions that are defined on the test model is that it is possible to abstract from distribution here since it is solved by the run time model that is responsible for supplying a consistent snapshot of the system's state. Thus, the constraints are defined given a synchronized system. Indeed, the synchronization needs to be provided by connecting the model with the actual system, as described later on. Listing 6.8 shows an excerpt of the constraint-based oracle used for the Hadoop case, that is a result of transforming the CCB to S#. The shown constraint describes, in parts, the requirement, that the parameters of the configuration need to be updated and are causing self-organization. The constraint checks three rules that imply an adaptation of the system, i.e., a response time outside of the specified slot and an exceedance of the budget. The constraints are formulated on the basis of the `YarnController` containing the necessary information (cf. Figure 6.12).

System and Environment Model The system model describes the components of the SuT, i.e., the *YARN* component as a domain model, that already a result of forming the KAOS model. This model is completed by the environment of the SuT having the environment model. The model-based testing paradigm pays off in this large-scale industrial case study due to its abstraction abilities making the approach scalable. For this purpose, the model must be focused on the test purpose, that is defined by the set of investigated requirements outlined before.

```
/* ... */
AdjustmentNeededConstraints = new List<Func<bool>>
{
    () => YarnController.AvgResponseTime > Model.HighResponseTimeValue ||
        YarnController.AvgResponseTime < Model.LowResponseTimeValue  ||
        YarnController.TotalServerCosts > Model.MaxBudget * 0.75
};
/* ... */
```

Listing 6.8. Partial S# component representing the constraint-based oracle in the Hadoop case study.

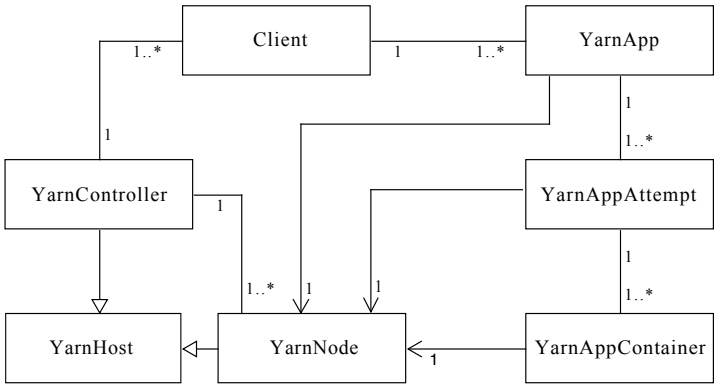


Figure 6.13. Graphical representation of a simplified version of the domain test model formed by the classes describing the SuT as well as its environment in the S# test model. `YarnHost` represents the basic class for all distributed components in the cluster. The `YarnNode` executes a `YarnApp` allocated by the `YarnController`. The `YarnController` is the adaptive part of Hadoop. The `Client` is the environment which is not controlled by the SuT. The `YarnController` is responsible for allocating a client's task (formulated as `YarnApps` that have different `YarnAppAttempts` stored in the `YarnAppContainer`) in the SuT.

Figure 6.13 shows the graphical representation of the classes that build the system and the environment model. In general, the Hadoop system follows a client-server-architecture which is reflected in our model: The environment of the SuT is formed by the client, which is the component which has the most influence on the SuT. Besides the client, the nodes and their connections to the *YARN* controller are also part of the environment, i.e., the controlled environment. This differentiation is of importance as the controlled environment is also used by the test oracle since parts of the functional requirements concern this control task. The other part of the environment, i.e., the client, is not controlled by the SuT, it is nevertheless interacting with the Hadoop system and driving the execution of the SuT, i.e., the tasks or requests sent by instances of the client class.

Test Model The test model serves mostly as a definition of the test suite. Hence, the test suite is described by two kinds of models: one for the controlled part of the environment of the SuT and one for the dependent environment, i.e., the client. The first part of the test suite is based on a fault-based test case description, the environment fault injection, as for the production cell. The latter part describes the environment as a probabilistic test model, able to deliver endless test inputs, the EPs.

Environment Profiles Environment Profiles are probabilistic models, that describe the interaction of the environment with the SuT. In the case of Hadoop, the interaction is focused on the client, that can submit tasks and consequently controls the workload. Testing the *YARN* controller demands a workload on the Hadoop system in order to activate it. The basic idea of the environment profiles is to generate test inputs that represent the most likely conditions. It is up to the test engineer to design a good environment profile for the test suite. For forming the EP for the Hadoop case, we use the most popular three benchmark collections that apply to Hadoop as well as information for empirical studies on the usage of Hadoop available in the literature [35, 42, 136]. These benchmarks are used for tweaking the parameters of *YARN*, among other things. The three benchmark collections are Hadoop MapReduce Examples [61], Intel's HiBench [79], and Statistical Workload Injector for MapReduce (SWIM) [34]. These have been clustered to extract the different possible tasks for Hadoop, resulting in 14 different types of actions that are grouped into four categories:

1. *Generators*
    - Text files: random text writer (rtw) and TestDFSIO -write (dfs-w)
    - Binary files: randomwriter (rw) and teragen (tgen)
2. *Data Processing*
    - Read: wordcount (wc) and TestDFSIO -read (dfs-r)
    - Sort: sort for text data and terasort (tsort) for binary data
    - Validate: testmapredsort (tstsort) and teravalidate (tval) for any sorting application
3. *Calculation*

- pi: Quasi-Monte Carlo method for calculating π
- pentomino (pent): solving the pentomino problem

4. *Simple Interaction* sleep and fail

The states shown are the 14 actions categorized above. Thus, a state change implies stopping one action (or completing it) and starting the next which is corresponding with the next state. After identifying these states, the transition probabilities have to be defined. In order to figure out these values, we analyzed the benchmark as well as other typical applications for Hadoop and the remarks of Zhang et al. [178] in detail. Further, we used the empirical analysis from the literature [35, 42, 136] to ground our numbers. The result is the transition matrix shown in Table 6.5 with the transition probabilities used in the environment profile.

Environment Fault Injection The second part of the test suite is formulated as environment faults. The faults are injected into the controlled environment of the *YARN* controller, i.e., the nodes and the connections between nodes, controller, and client.

Listing 6.9 shows the possible specification of two environment faults in S#. The component shown is a simplified version of the `YarnNode` class defines different attributes as well as functions of the node. The functions are used to represent the functionality of the component of the SuT and also for mapping the test model to the actual SuT for test automation. Since the node is part of the environment of the controller, we implemented different test cases in the form of environment faults. The activation is at random, which we extend in the next chapter of this thesis.

```
class YarnNode : Component {
  YarnController _connectedYarnController;
  bool _isActive;
  List<Query> _queries;

  public virtual void Activate()    {
      _isActive = true;
  }

  public virtual void AddQueries(List<Query> queriesToExecute) {
    _queries.AddRange(queriesToExecute);
  }

  [Transient]
  class ServerCannotActivate : Fault {
    public override void Activate() { }
  }

  [Persistent]
  class CannotExecuteQueries : Fault {
    public override void AddQueries(List<Query> queriesToExecute) { }
  }
  /* ... */
}
```

Listing 6.9. Simplified S# component representing a Hadoop `Node`.

	dfw	*rtw*	*tg*	*dfr*	*wc*	*rw*	*so*	*tsr*	*pi*	*pt*	*tms*	*tvl*	*sl*	*fl*
dfw	0.600	0.073	0	0.145	0	0	0	0	0.073	0.073	0	0	0.018	0.018
rtw	0.036	0.600	0	0	0.145	0.036	0.109	0	0.036	0	0	0	0.019	0.019
tg	0	0.036	0.600	0	0	0	0	0.255	0	0.073	0	0	0.018	0.018
dfr	0	0.073	0	0.600	0	0.036	0	0	0.145	0.109	0	0	0.018	0.019
wc	0.073	0.109	0	0	0.600	0	0.073	0	0.073	0.036	0	0	0.018	0.018
rw	0	0.073	0.073	0	0	0.600	0	0	0.109	0.109	0	0	0.018	0.018
so	0	0.073	0.036	0	0.073	0.036	0.600	0	0.073	0	0.073	0	0.018	0.018
tsr	0	0	0	0	0	0	0	0.600	0.109	0.073	0	0.182	0.018	0.018
pi	0.145	0.109	0	0	0	0	0	0	0.600	0.109	0	0	0.018	0.019
pt	0.109	0.109	0	0	0	0.073	0	0	0.073	0.600	0	0	0.018	0.018
tms	0	0.145	0	0	0	0.073	0	0	0.036	0.109	0.600	0	0.018	0.019
tvl	0.073	0.109	0	0	0	0	0	0	0.109	0.073	0	0.600	0.018	0.018
sl	0.167	0.167	0.167	0	0	0.167	0	0	0.167	0.167	0	0	0	0
fl	0.167	0.167	0.167	0	0	0.167	0	0	0.167	0.167	0	0	0	0

Table 6.5. Transition matrix of the environmental profile with the probabilities used in the test automation of the Hadoop system.

S# Testdriver for Hadoop

In order to fully automate the testing within S#, it is necessary to connect the SuT, here the Hadoop system, with the executable S# model. The connection is established by a test driver which is integrated into the S# code, written in C#. Two functionalities must be provided by the test driver, to enable test execution: (1) controlling the SuT by enabling the injection of faults in the controlled environment of the SuT and (2) monitoring the SuT with its controlled environment as well as the clients for the Hadoop system. Since the SuT and the test system are part of a distributed cluster, a connection between the test system and the SuT needs to be established. We use a *REST*-based test driver to execute the control commands and to gather information from the Hadoop system for monitoring. In order to keep the test system architecture unaffected from the concrete test driver implementation, the test driver is encapsulated in a particular interface. The primary function within the test driver implementation is to translate and transfer commands for controlling the SuT and to receive and translate monitoring information. The counterpart in the Hadoop system which is needed is the scripts used to supply the relevant functionality for controlling and monitoring.

Controlling the SuT The SuT is controlled by the test driver which is injecting faults into the controlled environment of the SuT, i.e., the activation of environment faults, and sends tasks to the Hadoop system. The later one can be directly generated at the test system via its interfaces supplied. The workload is generated by having function calls to Hadoop for the 14 different classes of actions (cf. Table 6.5). The functions make use of the workloads supplied by standard benchmarks we used for extracting the states of our EP. They are called from C# and thus directly executed from the test framework. Thus, it is, for example, possible to disable a network connection or to disable/shutdown a particular node of the Hadoop system as a fault activation.

Monitoring the SuT Monitoring is needed in order to update the run time model of S# after every step. The execution order of the steps is fixed and determined by the test engineer. In our case, we first updated the state of the model and afterward executed the selected test steps. Executing the steps is in the responsibility of the test driver as described before. For updating the model, we need to extract that information for the system as a snapshot. Gathering the information needed makes it necessary first to select which information is of concern. This step is already performed by generating the test model, here we selected the information needed. The system and the environment model are instantiated as a run time model. Thus, the information for its attributes has to be retrieved from the SuT that is available by on the one hand the Hadoop system itself (making use of the *graphite* extension, cf. Figure 6.12) and on the other hand the docker ecosystem. The data is retrieved by command line functions and needs to be extracted by a parser afterward. This parser is written in C# and maps the information into the S# model.

After the test driver is defined, the test engineer can abstract from these technical details and from synchronization by defining tests to be automated or the constraint-based oracle only on the consolidated model.

Case Study for Testing a Distributed, Adaptive Real World Software Systems—Research Question *R12* The experiences made by applying the Hadoop case study is summarized in the following, reflecting the abilities of the approach for testing an industrial SOAS, as challenged by research question *R12*.

Model-Based Testing of an Adaptive *Hadoop System* Application Work by Zhang et al. [178] showed that adaptive systems are not limited to artificial research case studies; they can be applied to a real-world application. In this evaluation, we showed that our concepts for MBT of SO mechanisms are also applicable for this particular real-world application: the self-adaptive controller of a Hadoop application. The central concept of the approach proposed in this thesis is to use run time models; the underlying model-based paradigm enables to handle the complex systems—here a distributed Hadoop system—by abstraction. Abstraction makes it easy to integrate the automated oracle, without worrying about the distribution of the system.

Implementing the Test Scaffold for Distributed Test Environment Indeed, the model-based paradigm made things easier by abstraction. However, test automation still somehow needs to cope with the complexity of the system when tests are executed and evaluated. This is done by the test driver that we integrated into our test system. The model defined which kind of information is needed to be extracted from the SuT and which information or actions needed to be executed on the SuT. The set of command line functions we defined under the hood of the test automation is still not as generic and as reusable as we would like it to have. It needs to be customized for each application by a test engineer. We showed that and how it is possible to do so for a complex real-world application. In future work, this challenge of connecting a complex system to the testing ecosystem in a generic way might be worthy of investigation, potentially using learning techniques.

Mutation Analysis of the Approach For the evaluation of the ability of the approach to detect failures, the universal mutator tool by Groce et al. [69] has been used to alter the Java-based implementation of the SO mechanism. The universal mutator can modify a given code fragment in order to introduce the standard mutant operators known from the literature. After mutating the code, different versions of the original class are supplied that are compilable, i.e., survive first a smoke test. The results are 43 different versions of the SOuT. These versions have been hand selected by they aim of choosing the most diverse mutation settings. The diversity was analyzed statically, also by hand, by comparing the effect of the mutants on the mechanism of the SO algorithm.

One test run took on average 4 hours and executed 290 test cases. The number of test cases to be executed was set based on the experiences made by previous test runs

since that number leads on average to an execution of a pair-wise combination of the specified environment faults and a statement coverage of the EPs. However, the random combination of the test cases leads to 14% of the test runs that are stopped before the 290 test cases are achieved, since no more valid system configuration was possible for the SOuT in that case. The number of test cases executed is rather low, compared with the numbers in the case studies where the SO mechanisms are extracted from the systems. That is not the case in this evaluation where the SO mechanism operates in its natural habitat, i.e., in full integration. Nevertheless, it was possible to reveal the different mutants in the approach proposed in this thesis. That demonstrates the ability to find standard mutation faults in the code. However, there might be unique SO faults, like the ones shown in the evaluations before, that are not evaluated here. There is a need for defining standard mutation operators for SO mechanisms in future work.

Summary and Outlook. Testing aims at revealing failures from the SuT by executing it with different inputs. For this purpose, it is necessary for the test engineer to gain an understanding of the intended behavior of the SuT. Thus, it is necessary to form the requirements for testing and derive a test suite. This mental model of the test engineer is made explicit in MBT. Model-Based Testing is used for structuring the testing approach and its activities by allowing for automation of testing activities. This allows the test engineer to shift her focus on design and analysis of testing rather than on implementation and execution. For testing complex systems this focus is needed. We showed in this chapter how the paradigm of MBT is applied for testing SO mechanisms. However, the static concept of MBT is not able to cope with the dynamic and adaptive behavior of SO mechanisms. Thus, MBT was extended in this chapter to cover the needs for testing SO mechanisms. The main presented contributions in this chapter are:

1. The concept of closed-loop MBT has been introduced to be able to feedback the information of the executed SuT and the executed tests into the model.
2. Model reflections for MBT was presented as a concept for enabling closed-loop MBT in a run time model.
3. The concept of executable models for testing was proposed, allowing for test scaffolding of SO mechanisms.
4. A method for modeling continuous SO mechanisms within a probabilistic test model was introduced. This enables to describe the ever-changing environment of an SO mechanism and allows for describing abstract test cases that are automatically instantiated by executing the model.
5. A method for modeling discrete SO mechanisms was introduced. The model extends the concepts of fault-based testing toward allowing to describe abstract test cases for discrete SO mechanisms. This concept is integrated into the concept of the executable test model, allowing for automatic test cases instantiation.
6. Back-to-Back testing was proposed for SO mechanisms. We showed the special need for this concept in the context of SO mechanisms. The autonomy for the SO mechanisms is designed by giving the mechanisms space for decisions. That challenges the specification of the test system and the SuT. Back-to-Back testing enables to provide a tool for testing that special concern.
7. A thorough evaluation was delivered showing the strength of the proposed concepts. The evaluation was carried out on five different case studies. The results showed the success of the proposed MBT concept.

The concept for MBT allows for fully automated testing once the model has been created. The model-based paradigm shows its strength especially in its abilities for abstraction, making it possible to handle complex classes of SuTs. Generating actual tests within the model is done by executing the model. So far, the execution and generation was random, showing already the abilities of the approach to reveal failures. Within the next chapter, we will elaborate techniques for test case selection. The test cases are selected in order to reveal failures as fast as possible. This is of high importance, especially when a test end criterion has to be defined. The test end was set to a more or less random number. We will elaborate next how this is done more systematically.

Summary. The testing of SO mechanisms is demanding on different levels, one challenge, addressed in this chapter, is to handle the flat-branching state space of the SuT. Common test case selection techniques are designed for deep-branching state space, making these techniques less useful for selecting test cases for SO mechanisms. We will show how to apply the concept of boundary-interior testing to SO mechanisms and how to use a search-based testing approach for online test case selection. Further, an approach for more flexible, adaptive test cases is presented in this chapter, that can react at run time to changes of the SuT. The evaluation will show how these techniques are able to speed-up failure detection in the MBT setting presented in the chapter before. The content and contributions of this chapter are published in [51, 53–55, 133].

7 Test Case Generation for Flat-Branching Test Problems

7.1 Related Work **135**
7.1.1 Search-Based Test Case Generation 135
7.1.2 Adaptive Test Automation 136
7.2 Boundaries of Self-Organization Mechanisms: A Boundary-Interior Test Case Generation Approach **136**
7.2.1 Boundary Interior Test Case Generation for SO Mechanisms via Search-Based Testing 138
7.2.2 Heuristic-Based Selection Strategy for Automated Online Test Case Selection and Reduction 139
7.3 Adaptive Test Cases to Enable Reasoning During Test Execution **140**
7.3.1 Annotating the Purpose a Test Case for Enabling Self-Reflection . . . 141
7.3.2 Outlook: Planning Optimal Rule Instantiations by Optimizing Diversity of the Test Cases 143
7.4 Evaluation **144**
7.4.1 Production Cell—Boundary-Interior Test Case Generation 144
7.4.2 Load-Balancing Web-Service—Adaptive Test Case Execution 148

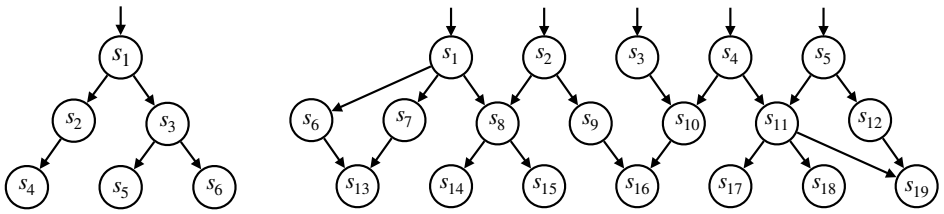
In the previous section, a thorough approach for Model-Based Testing (MBT) of Self-Organization (SO) mechanisms has been presented. The evaluation results showed the capability of the approach to address the particular requirements for testing SO mechanisms. These are described in Chapter 4 as follows:

1. Non-Determinism in the execution of the SO mechanisms, due to the used SO algorithms (e.g., particle swarm optimizer).
2. Ever-changing environment that is unpredictable due to its complexity that is determining the action and solution space of the SO mechanism.
3. The Self-Organizing, Adaptive System (SOAS)'s components are interacting intensively.
4. Concurrent execution within the controlled system and, possibly, concurrent SO mechanisms.

However, the complexity of the test model, which can use reflection to adapt to an System under Test (SuT) changing at run time, and the complexity of the Self-Organization Mechanism under Test (SOuT), that is adapting itself as well, is leading to many situations to be investigated by test cases. The state space of the test setting is vast and complicated to exploit. We have many different configurations of the systems, which are possible to start from and even more different actions and combinations of actions (that depend on various conditions) to continue. The interesting aspect is, that the state space is, not as common for testing problems [17, 121], rather flat-branching instead of deep-branching. Figure 7.1 is illustrating the difference between the flat and deep branching state space: Figure 7.1a shows the state space as it is common in software testing, having one initial configuration as a starting point and branching depth (especially for dissolving loops and returns). For SO mechanisms' state space there are different configurations and starting points given leading to many different actions afterward. This is resulting in a flat branching state space, as shown in Figure 7.1b. The structure of the state space of SO mechanisms is a consequence of the non-determinism in the SO algorithm used for computing new system configurations. However, we showed in the previous chapter, that it is possible to use the test model for generating test cases for testing by simply executing it; resulting in random tests. Therefore, the concrete environment faults to be activated and transitions in the Environment Profile (EP) to take are chosen randomly. Known test selection strategies, e.g., modified decision condition coverage (cf. [121]), are, as stated before, designed for a rather deep-branching state space, formed by static propositional statements in the code. For SO mechanisms, having a flat-branched and hard to predict state space, that can change at run time, these techniques are hard to apply. One interesting observation is, that branch coverage is for instance achieved in every reconfiguration for the SO mechanism in the energy grid case study. However, the evaluations in Chapter 6 haven shown, that a single reconfiguration is not adequate for thorough testing. We showed that it is possible to use the coverage on the test model, e.g., the states of the EP, as an alternative criterion. Nevertheless, the test selection criteria used in most approaches are focused on standard adequacy criteria and thus not applicable here.

In this chapter, we introduce two approaches, which can exploit the unique characteristics of the state space of SO mechanisms. These are used for improving the test case generation compared to the results which have been achieved by random testing.

The boundaries of SO, i.e., the state where barely a new configuration is possible for an environment situation, turned out to be error-prone states for SO mechanisms, as we will elaborate in the evaluation of this section. The following fact explains that: the Corridor of Correct Behavior (CCB) implicitly defines the border, and the concrete implementation of the SO depends on this and most failures are likely when only less correct options are given. We will show how techniques of search-based testing are used for test case generation. The search algorithm is directed toward the borders of the SO and directly execute the explored state as a test case. We will show how heuristics can improve a breadth-first search, making use of the assumption of the flat-branching state space. The results are shown in the evaluation with the production cell case study.



(a) A deep branching state space, as common for testing problems, starting at s_1 with one initial starting point and branching rather deep than flat. Note that this tree structure is demanding for unrolling loops and returns, leading to deep branches.

(b) A flat branching state space is emerging from different initial starting points, as common for SOuT.

Figure 7.1. The state space is in testing mostly converted into a tree structure for illustration and for applying test case generation resp. selection strategies as well as for applying adequacy criteria. The structure of this tree is common to be deep branching (cf. [17, 121]), as shown in Figure 7.1a. For SO mechanisms, the state space is different: it is more flat branching, as shown in Figure 7.1b. This is due to the multiple different possible configurations of the system to start from and the decisions to take afterward.

Further, we introduce a concept of adaptive test cases, that is suited for testing adaptive resp. SO mechanisms. To put it into a nutshell, the test cases are defined not in a deterministic and inflexible way as common for automated test cases, but more in a flexible way by using the run time information of the model. That enables the definition of the intention of the test case based on run time situation, allowing to execute the test cases in different possible shapes.

7.1 Related Work

Test case selection and generation is a field of intense research, as shown in the survey of Anand et al. [5]. The techniques introduced in this chapter are building upon the MBT approach of Chapter 6. We investigate the use of search-based test case generation for effectively generating tests from the model, and we extend the automation abilities to gain adaptiveness in the test execution. The related work classifies these two areas.

7.1.1 Search-Based Test Case Generation

Model-based and search-based techniques, among other things, are used to cope with the challenges of generating the most promising test cases for execution. The concepts of search-based software testing, as summarized by McMinn [98], are making use of metaheuristic optimizing search techniques for directed test case generation. The target function is giving the direction within the search landscape. Miller and Spooner [102] have described this test concept first, making use of optimization techniques for gen-

erating test inputs maximizing code coverage. The critical parameter of this test case generation approach is to select the search target and describe the fitness landscape. The challenges are to encode the testing problem adequately and in a suitable form for the optimization tools. The encoding is appropriate if the resulting fitness landscape provides clear guidance, i.e., without plateaus. Different approaches [83, 101, 168] exploit branch coverage within the control-flow of a program and discuss various forms of the fitness landscape. However, they are not adequately describing the problem of SO mechanisms, as outlined before. Fraser et al. [62] describe a further approach for search-based testing using the search mechanism of a model checker. We pick up these thoughts and implement a breadth-first algorithm for our search-based approach. Further, we have to design new optimization functions that reveal failures in SO mechanisms. To my knowledge, there is no other online test case selection approach that could be applied to this class of SuT.

7.1.2 Adaptive Test Automation

The here proposed approach for adaptive test automation extends current test automation concepts by a notion of adaptiveness (w.r.t. system and context states). Self-aware test models enable us to design a new kind of automatable test cases that incorporate situational aspects and the purpose of the test case, but also information about the correct system state. As can be seen in Polo et al. [124], current approaches primarily execute test scripts, mostly without any context description, or replay captured scenarios that have been recorded through manual testing. Those approaches have a significant capability in fast and efficient execution, but lack of maintainability and have to be reworked after changes in the system, and further have to be governed with high effort. Though they strive to optimize effectiveness in test execution, this is often at the cost of reactiveness and adaptiveness in regards to changing contexts or system states. There is, however, a need for reactive tests—especially when dealing with systems that can change their internal structure in response to contextual changes for themselves; or in other words: *adaptive systems need adaptive tests*. We solve this by using models as run time reflection of the current state. The model@runtime community inspires this approach (cf. Aßmann et al. [12]), even if they are concerned with adapting system strategies instead of testing. Existing approaches for testing adaptive systems are focused on test case generation and the usage of the test output to tweak the system performance (cf. Siqueira et al. [148]). The automation of test suites was previously only done for dedicated test cases, but not in a general approach as we propose it.

7.2 Boundaries of Self-Organization Mechanisms: A Boundary-Interior Test Case Generation Approach

A huge state space is a common challenge for software testing [17], but SO mechanisms add a further dimension. Most approaches in software testing coping with a vast state space make use of its structure to reduce the number of test cases that need to be executed. For instance, an infinite loop in a code fragment means an infinite state space, but its ramification degree is rather small. A mechanism applied here is boundary-

interior testing [121] that cuts deep branches at specific lengths. Self-Organization mechanisms, however, are mostly based on heuristics for coping with the ever-changing environment, making the result non-deterministic. This characteristic leads to a broad and somewhat flat branching structure of the state space and makes most of the classical techniques hardly applicable directly. We present an approach called boundary-interior testing for SO mechanisms that can cope with this flat branching structure. Unlike standard boundary-interior testing, the approach is not to rely on the deep branching state space, but on the system structure that is controlled by the SO mechanism. The approach reflects the fact that SO mechanisms mostly rely on the structure and the degrees of freedom that could be used for self-organization. In the production cell (cf. Chapter 2), the robots and carts, their abilities, and their role allocations define the structure for the SO mechanism. The different possible role allocations to achieve a given task define the degree of freedom. This structure is the input, as well as the output of the SO mechanism. Similar to classical testing approaches, e.g., boundary value testing, we rely on empirical knowledge concerning the failure distribution. The boundaries where most failures are revealed are at the boundaries of SO mechanisms where the structure hardly allows further reconfiguration of the controlled system.

This insight answers the most crucial question for test cases prioritization and generation: What kind of situations should be tested first? Since testing cannot even come close to covering the complete state space of SO mechanisms answering this question is of utmost importance as we would like to reveal as many failures as possible within the testing period. The characteristics of the SO mechanism also affect the way tests can be executed within a test harness. Because SO algorithms mostly rely on heuristics and machine-learning techniques, the environment of the SO mechanism plays a decisive role: SO mechanisms are not testable by providing simple input values, running tests, and gaining an output. Instead, a realistic environment setting must be supplied as a test harness (as already discussed in detail in Chapter 5). To summarize, we are faced with two questions for test case generation:

1. What kind of situations are the most error-prone and should be tested first?
2. How can test setup and test case selection be used to reach these situations fast?

Boundaries of SO Mechanisms within the Corridor Enforcing Infrastructure (CEI) Figure 7.2 represents different test cases generation and selection strategies to address these questions. Since we present an online test case generation procedure, test case generation and selection goes along with each other. All possible test cases (resp. the complete state space of the program) are represented as dashed lines. The x-axis represents different configurations of a system and the y-axis different test cases. Exhaustive testing is achived by using a depth-first search approach and going through every possible case, shown on the left side of the figure. Executing every possible test case is often undesirable or even impossible. Thus, we test with a focus on situations where we expect a solution from the SO mechanism. We can cut off test cases where it is impossible to reorganize the system, as done in the middle box of Figure 7.2. The line that is drawn in Figure 7.2 shows the boundary between possible and impossible reorganizations, the upper part represents all test cases where no reorganization is possible, where it is

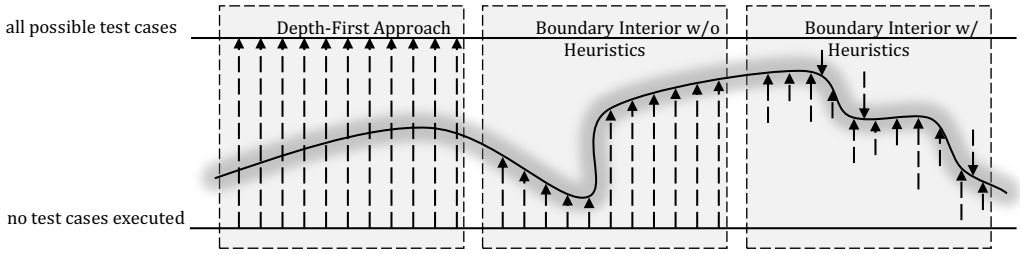


Figure 7.2. The three boxes represent different test case generation strategies. The x-axis of the graphic shows different system states for a particular SOAS, which are formed by the different possible configurations. The line shows the boundaries of the mechanisms for this SOAS. Test cases are shown as small black lines. We assume the most failures to occur in the shaded areas around the boundaries.

enough to select a few negative tests. Thus, the answer is to test inside the boundaries of SO. However, most of the failures (in our evaluations even all failures we revealed) are not only inside the boundaries. They are close to the boundaries. This insight leads to the next possible test case reduction, shown in Figure 7.2 at the right: we select test cases with the highest priority that are very close to the boundaries.

The intuition of the approach is to reach the boundaries by reducing the redundancies within the system since SO's abilities build upon redundancies. With many redundancies, it is harder to reveal a failure since even a random algorithm for SO (which is not correct in every case) has a higher chance to return the correct result (thus, no failure is revealed even though there is an error). The chance increases if there are less correct solutions to choose from, as it is the case at the boundaries.

7.2.1 Boundary Interior Test Case Generation for SO Mechanisms via Search-Based Testing

For test case generation within one configuration of the system, we adopt the concepts of boundary interior testing, where the idea is to select test cases at the boundary of expected behavior changes. The boundaries of SO mechanisms are states of the system where only a few solutions for reconfiguration are given. Reconfigurations, as we consider them, are mainly driven by changing environmental conditions that force the system to reorganize itself. In our test model, we define these changing conditions as environment faults or environment behavior of the controlled system such as a robot being unable to apply its tools in the production cell case study.

To find the environment faults that bring the system to its boundaries, we use a search algorithm that implements breadth-first search on the fault activations. Similar, the environment behaviors could also be searched in this fashion, however, as the model allows infinite depth we have to restrict it to a certain degree. The standard approach is to check the fault sets by increasing cardinality. Thus the approach also includes test cases for inner boundary tests. The concepts of boundary interior testing for SO mechanism is exemplified in Figure 7.2. The left box refers to a naïve approach where test cases are selected in a depth-first attempt, and the boundaries of the SO mechanisms are

not taken into account. That implies that many negative test cases are executed where fewer faults are expected to be revealed; in our evaluation, no fault has been detected by these negative tests. The middle box of Figure 7.2 is representing the search-based approach for boundary interior testing and covers the interiors and boundaries. The right box shows an extension where only the boundaries are considered by conducting the breadth-first search algorithm with different heuristics that allow for skipping states. That is done by selecting component fault sets first where more faults of the same kind are activated, and subsumption relations between component faults are exploited.

7.2.2 Heuristic-Based Selection Strategy for Automated Online Test Case Selection and Reduction

The proposed breadth-first search algorithm is used for searching the boundary of SO as described. This search strategy is illustrated in Figure 7.3, the search starts with the smallest set, here a set of environment faults, the empty set. Whenever an environment fault set is identified to reach the boundary of SO all its supersets are also known to be at or beyond the boundary and consequently not included into the test suite -– following the boundary-interior testing concept. When on the other hand an environment fault set is not at the boundary, i.e., a reconfiguration of the system is possible to fulfill the system's goals, its direct supersets are examined in the next round. Consequently, all subsets of large sets at the boundary are examined. Using heuristics, we can attempt to identify these fault sets sooner and thus avoid testing their subsets, i.e., executing tests right around the boundary of SO. Next, we are investigating two heuristics used in the case of environment faults.

Fault Subsumption Heuristic In many cases, a model has more severe environment faults that subsume other, less severe ones. In the production cell case study for example if a robot breaks down completely, it does not matter if any of its tools are faulty as well — that failure subsumes the other ones. Since the relationship cannot be inferred automatically without checking all environment fault sets first, it must be declared in the model. The heuristic makes use of that relation by suggesting test cases closer to the boundary: Given a test case—a set of component faults—that allows for further reconfigurations and does not reveal a fault in the implementation, it adds all environment failures subsumed by the test case. Since only subsumed faults are added, the resulting test case should still be inside the boundary, but much closer to it.

Minimal Redundancy Heuristic When analyzing the reconfiguration limits, there are two relevant aspects: the robots' available tools and the possible routes between them established by the carts. Relevant environment faults (as test cases) are those that imply a loss of capabilities or limitations to the routes between the robots. An environment fault set right at the boundary of the reorganization is one that leaves just enough of the system's functionality intact to allow for reconfiguration. Figure 7.3 shows an incomplete Hasse Diagram that shows some possible test cases (formed by different combinations of fault activations, i.e., Robot 0 fails entirely (r^0), Robot 1 fails either

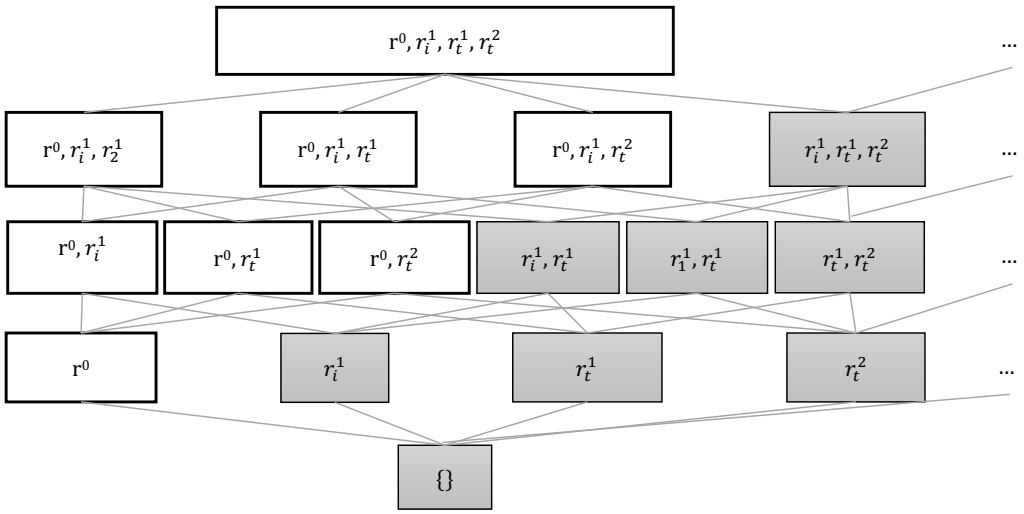


Figure 7.3. This Hasse Diagram demonstrates some of the different combinations of fault activations.

with the inserting tool (r_i^1) or the tighten tool (r_t^1), and robot 2 fails with the tighten tool (r_t^2). In the assumed setting, the robot 0 is the only one with a drilling tool and so necessary for a task that includes drilling. Removing the drilling tool by activating the fault would lead to a situation where no reconfiguration is longer possible, i.e., a situation outside the boundary. In Figure 7.3, the grey test cases are inside the boundary and the white ones outside. The proposed heuristic chooses situations as follows: fault sets are selected in a way that a maximum number of faults are activated so that the given task could be still fulfilled (for the sake of simplicity, the carts in our case study are not considered). A corresponding choice would be for example given with the task drill, insert, and tighten the fault set r_i^1, r_t^1, r_t^2: In the resulting situation, robot 0 has to carry out the drill, insert, and tighten, and there is no other correct solution. Thus, that fault set is next to the boundary, and all subsets do not have to be tested. Indeed, it could not be guaranteed that no failures could be revealed by applying tests that are formed by a subset. However, due to the characteristics of SO mechanisms, it is unlikely.

7.3 Adaptive Test Cases to Enable Reasoning During Test Execution

An insight gained from the search-based test case generation is that the current state of the system is determining the decision for the next test execution. That is possible, as the search is executed on the model, that is a run time model reflecting the current state of the system. The use of the current state of the system is enabling the test generation to be directed toward the boundaries of SO. Besides the boundaries of SO there might be other, more system-specific test goals, that a test engineer can define by test cases. However, to enable these test cases to be automated, it is necessary to define their intention in the run time context. The described test automation approach enables the definition of the intention of the test case. The main constituent of the test automation is the adaptive test execution which is based on the self-awareness originated from

```
class Server : Component {
  Proxy _connectedProxy;
  bool _isServerActive;
  List<Query> _executingQueries;

  public void Activate()    {
      _isServerActive = true;
  }

  public void AddQueries(List<Query> queriesToExecute) {
    _executingQueries.AddRange(queriesToExecute);
  }

  [Activation("TooFewServers", selectedServer="2")]
  [Persistent]
  class CannotExecuteQueries : Fault {
    public void AddQueries(List<Query> queriesToExecute) { }
  }
  /* ... */
}
```

Listing 7.1. Partial S# component representing a server of load-balancing web-service system ZNN.com.

the run time model established in Chapter 6. Context patterns define the test strategy, i.e., a description of an environment state, that is denoting the intended situation of the abstract test case. An instantiation strategy further completes the context pattern. Each abstract test case, which is what should be defined by the test engineer in this approach, e.g., by environment faults or environment behaviors, has to be instantiated for execution. The instantiation is the provision of data that is needed for the parameters of a test. Let us consider an example from the load-balancing web-service case study in Listing 7.1: The test case is described by the environment fault `CannotExecuteQueries`. This test case is abstract, as it is left open which servers are affected by and activation a when the test case should be executed. A possible concrete instance of this test case contains a run time situation and information for the concrete servers to fail. As the information for the concrete instance needs information from the run time of the test, this information is defined based on the run time model, by a pattern for activation, here, `TooFewServers` is a pointer on a boolean function, and an option for the selection of servers, here set to `auto`, i.e., letting the test automation make the choice.

In this way, it is possible to define different adaptation rules for the test execution within the model that trigger which of the anticipated behaviors of the environment to be simulated and which faults to be activated.

7.3.1 Annotating the Purpose a Test Case for Enabling Self-Reflection

Having feedback from the test execution, as presented in Chapter 6, is enabling to access the run time information during test execution for test case selection. Thus, the purpose of the test case can be defined for its automation, as proposed in this section. The defined purpose enables an advanced test case selection: the test cases can be suited for different possible run time situations and are possibly instantiated in different shapes, depending on the current test situation. The situation awareness is delivered by the

```
List<Func<bool>> CriterionTooFewServers = new List<Func<bool>>
            {
                () => Proxy.ConnectedServers.Count < 3,
                () => Proxy.ConnectedServers.Count(s => s.IsServerDead) <
                    Proxy.ConnectedServers.Count
            };
```

Listing 7.2. Criterion dependent activation condition for `TooFewServers`

model reflection and the run time model concepts, shown in Chapter 6. The adaptation of the test suite to the situation is enabled by defining the test case's purpose. For the test case, described as an environment fault, shown in Listing 7.1, the purpose is named as follows: The test case is designed for a situation with too few servers that are available in the system. In this situation servers should no longer be able to execute queries, testing the reaction of the SO mechanism in this extreme situation. A tag here summarizes the purpose: `TooFewServers`. This tag needs to be linked with a more detailed description of the situation, defined on the run time model. In this case, in the S# context, it is implemented by a boolean function that is linked by the naming. Every activation criterion, which is the name of the criterion tag, has to offer such a function. The function is stating whether or not, depending on the current instance of the test model, the test should be executed. In the current situation, a very simple functional description is listed in Listing 7.2.

Next, to the criterion, there is an additional criterion, that is necessary for instantiation: the information for which instance of the test model this test cases should be instantiated and executed. In Listing 7.1, the `selectedServer` property is set to 2, whereby the server is replaced by any other possible class within the test model that is containing test cases. Here, it is possible to annotate the server count. If for example `selectedServer=2` is set, always two instances are picked at random for instantiation.

For the case, that environment profiles describe the test suite it is almost the same approach. We define a criterion for each transition, it is a guard, the guard states whether or not a transition into another state is allowed, for all outgoing transitions that are permitted the new probability is computed, so that the sum of all outgoing transitions (self-transition included) is equal to 1 and the ratio is the same as before. Let us consider a simple example: A state has three outgoing transitions with the probability $0.5, 0.3, 0.2$. Ff the outgoing transition with the probability 0.5 is not available the updated value is calculated by $0.3 + (\frac{0.5 \cot 0.3}{0.3+0.2}) = 0.6$ for the first value and $0.2 + (\frac{0.5 \cot 0.2}{0.3+0.2}) = 0.4$ for the second.

Thus, it is possible to intervene in the instantiation of the abstract test cases definition, as introduced in Chapter 6. This possible intervention is used for the reasoning in the test case selection procedure to evaluate the criteria of the test suite. Thus, the test case selection is directed by the test engineer with support from the test automation. The incorporated run time information in the test suite enables the automation to adapt to the SuT.

7.3.2 Outlook: Planning Optimal Rule Instantiations by Optimizing Diversity of the Test Cases

If the criterion for the instantiation is not set to a concrete number, but to `auto`, a learning-based approach is in charge of this decision. Depending on the chosen system configuration, a particular fulfilled condition may subsume quite some concrete implementations and span a vast space of optional steps to execute at specific states. The question, which of them to choose, i.e., which test step to execute if there are many, resembles the challenge of instantiating logical test cases by concrete ones in traditional testing. Let us, for instance, consider the criterion which introduces the persistent fault `CannotExecuteQueries` in listing 7.1. During the execution of the test, this environment fault needs to be instantiated for one out of all the servers which are deployed in the considered configuration. The demand for an instantiation results in a decision process at run time.

The Need for a Planning Module

Typically, substantial environmental state spaces prevent us from demanding manual solutions for resulting decision processes from the test engineer. Thus, we complete the test automation with a planning module that can automatically solve this job. In the aforementioned example this module is activated by the keyword `auto` at activation `TooFewServers`. This keyword marks the choice of which server to be affected as being non-deterministic and thus to be decided by the planning module.

Parametrized with some goals, which we assume to be given by the test engineer, the planning module uses the executable models to search for optimal decisions concerning the instantiations. A worthy goal is a kind of *action diversity*, which we see as a counterpart of code coverage criterion in the context of SOAS. This test indicator can be used to measure the difference between the system traces which are expected to be triggered by a particular rule instantiation. Maximizing the action diversity thus means to maximize the difference of triggered traces.

Learning the Distance Metric

As shown by Reichstaller and Knapp [132], the underlying behavioral distance function can be learned on-the-fly by simply observing the SuT in the simulation with a so-called Diversity Optimizing Learner (DOL). It is necessary to define diversity within the current test model, to enable the DOL. So, the run time test model (of the SuT) is to define the components and properties that described diversity. A diverse situation in the load-balancing web-service case study is defined by the number of dedicated servers that are affected by an action carried out by the proxy component, i.e., a situation where activation or deactivation involve only one server is diverse from a situation where four are concerned if there are six servers in total.

Besides the set of environment faults resp. environment behaviors based test cases that are supplied by the test engineer a machine learning approach enables to complete resp.

evolve the test suite by a diversity optimized learner. The evolvement of the test suite is focused on the environment-based test inputs and can be conducted in two modes:

1. the evolved test cases amend the existing ones by test cases that are resulting in a very similar test situation of SuT (*low diversity*) and
2. the evolved test case is very diverse in the resulting test situation compared to the existing ones (*high diversity*).

7.4 Evaluation

The evaluation for this chapter is two folded:

First, we are investigating the gains of the boundary interior test case generation approach where different search-based testing techniques are applied. For this purpose, the production cell case study is selected. In the previous chapter (Chapter 6), this case study was used for the evaluation of the MBT approach. The test case generation techniques in this chapter are based on this approach. The result of random testing is now compared to the directed, search-based test input generation techniques provided in this chapter.

Second, we choose the load-balancing web-service. Here, we will show how the concept of adaptive test cases is applied and what extensions are necessary for the model developed in Chapter 6. The emphasis on the evaluation is on discussing the new situations that are investigated by having an adaptive test suite.

7.4.1 Production Cell—Boundary-Interior Test Case Generation

For the evaluation we focus on the following research questions:

RQ 1: Are we able to reveal the same failures using our heuristic-enabled boundary-interior approach as with an exhaustive testing approach? Which cases can we possibly miss, and which do we miss in our case study?

RQ 2: How much can we improve the testing performance?

RQ 3: Can the hypothesis that failing test cases are located near the boundary be validated?

RQ 1: We tested a S# model of the production cell case study. In Chapter 6, we previously evaluated our MBT approach instantiating the test at random using the production cell case study. We used an SO mechanism based on the MiniZinc constraint solver and revealed several failures. For this evaluation, the same faults, as the one revealed in Chapter 6, were injected into the C#-based implementation producing the following failures:[1]

[1]Note that the phase of the SO mechanism, as described in Chapter 5, where the fault that is causing the failure is located is annotated with Detection, Computation, and Distribution.

F1: (Computation) Transitive connections between robots, i.e., connections that involve several carts and intermediate robots, were interpreted as direct routes. When trying to find a cart capable of transporting workpieces along these routes, the algorithm failed to find any.

F2: (Computation) The controller considered routes to be unidirectional, but they were meant to be bidirectional. As a result, the controller overlooked solutions, leading to incongruities with the test oracle.

F3: This failure, was too specific to the MiniZinc-based implementation and was thus not reproduced.

F4: (Distribution) The system model was misinterpreted regarding the pre- and post-condition of a role allocation. The condition should describe the capabilities already applied, but instead contained the remaining capabilities that were not yet applied.

F6: (Computation) The SO algorithm restricted the concatenated length of roles for a task. This restriction resulted in missed solutions when workpieces had to be transported between many different robots, necessitating a large number of roles that are transport-only.

F7: (Detection) The observer was missing a constraint, namely the I/O-Consistency (cf. [5]). When a cart was deactivated which transported workpieces between robots, this was not detected. Consequently, robots attempted to work on workpieces that were not where they were expected to be.

F8: (Distribution) Besides the failures revealed in Chapter 6, an additional pre-existing failure was discovered. When working with transitive routes, intermediate robots should be assigned a role meant only to receive a workpiece from one cart, and to pass it on unmodified to the next cart. These roles were not assigned to the intermediate robots, which should fulfill them, but to the last robot receiving the workpiece.

We analyzed seven different system setups, with varying modes of analysis: the depth-first approach, boundary-interior testing with the search-based testing approach presented in this chapter, and the combined approach using boundary-interior testing and heuristics. Each failure was analyzed individually. Since we strive to detect implementation faults in the model, we terminated the analysis of a model as soon as a failure manifested. The failures discovered during these analyses manifested themselves in different ways:

- **Exceptions:** Faults can cause the model to reach an inconsistent state, resulting in exceptions. For instance, *F1* caused an exception during reconfiguration, because no cart with a matching route was found. Thus, the reconfiguration never completed. *F6* also caused exceptions, but during production instead of reconfiguration, because workpieces were missing.
- **Conflicts with the test oracle:** In some cases, the reconfiguration algorithm might miss solutions and consider the current configuration problem to be unsolvable, even though the test oracle considers it solvable. *F2* and *F5* resulted in

such cases. Although not present in our evaluation, it is also conceivable that the controller finds a solution when the oracle considers the problem unsolvable.

- **Incorrect solutions rejected by invariant check:** The fault leading to failure *F7* caused the controller to compute an inconsistent configuration. A subsequent check of the invariant will detect it and reject the configuration.
- **Not at all:** Some faults did not manifest during the evaluation. For instance, *F4* resulted in configurations that were technically incorrect, but whose flaws did not affect the system nor were they detected by an invariant check.

There are specific links between the phases of an SO mechanism, as described in Chapter 5, affected by a fault and the way it manifests. Faults in Detection can only prevent necessary reconfigurations or prompt unnecessary ones. Similarly, faults in Distribution cannot provoke conflicts with the test oracle since distribution only occurs after the controller found a solution and checked for incongruences with the oracle. The most important result when comparing the testing approaches is that in our case study boundary-interior testing and the heuristic-based approach detected all failures detected by the depth-first approach as well, in all seven models we analyzed.

RQ 2: The largest system model we analyzed is illustrated in Figure 7.4. Table 7.1 shows the number of test cases necessary to detect each fault in this model and the time required to execute them. The performance results here are two-fold: For failures *F2* and *F6*, there is no significant improvement. This fact is acceptable, as they require very few test cases and very little time even with the depth-first approach. For the other failures, however, the speedup when using the combined approach is substantial. The combined approach also clearly improves upon boundary-interior testing without heuristics. The number of test cases decreases more than the required time.

RQ 3: The reasoning for applying the heuristics to the test selection problem was the supposition that the failing test cases would be located close to the boundary. The model shown in Figure 7.4 was further investigated, to validate this hypothesis. All detected failures were analyzed individually; using the depth-first approach, without stopping the analysis after the first failure was detected. The failing test cases were compared to the sets found by a complete analysis without any failures. This exhaustive test revealed that all these 67 failing test cases were located just below the boundary, thus validating the hypothesis. Since the heuristics were explicitly designed to find large sets of failures to be activated, just below the boundary, this explains the observed performance gains. We also investigated the question of error masking. For this purpose, we began with the seven different system configurations, each containing all the injected faults. After each analysis, we removed the detected faults from the models and re-ran the analysis. In this fashion, we were able to detect all the failures we could detect in isolation: The first analysis detected *F2*, the second *F1* and *F6*, and the last *F5* and *F7*. *F4* of course remained undetected.

At first glance, the results indicate that the proposed approach for test case generation does not pay off as expected in most cases. That is mainly an effect of the kinds of faults

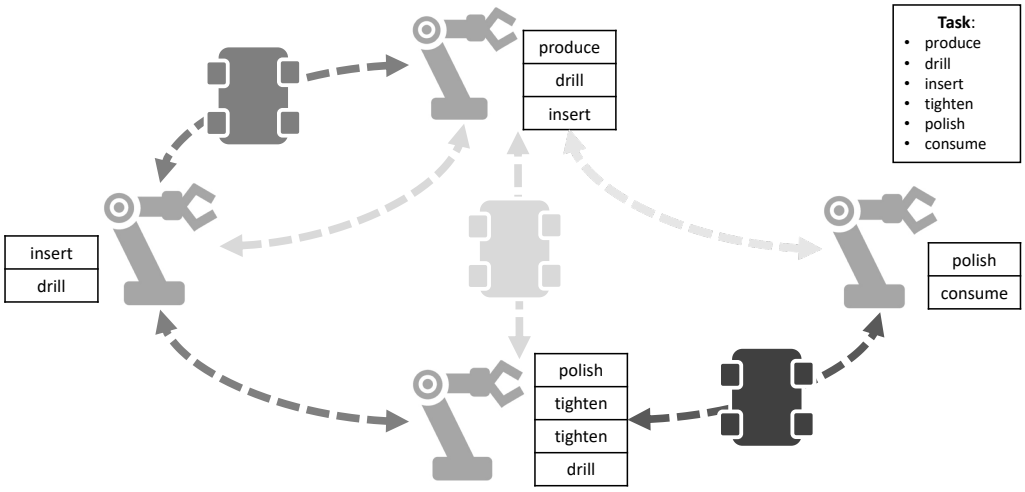


Figure 7.4. This diagram illustrates one of the production cell models we analyzed. There are four robots, each annotated with their available tools. They are connected by three different carts, whose routes are shown in the respective shades of gray. The production cell processes workpieces according to the task seen in the upper right corner.

	Depth-First		Boundary-Interior			Heuristics			
Failure	time	#test cases	time	#test cases	speedup	time	#test cases	accuracy	speedup
F1	5.2s	353	3.7s	212	138%	2.9s	187	34.00%	176%
F2	1.0s	27	1.0s	27	138%	1.0s	55	3.03%	100%
F5	45.8s	8662	24.8s	1566	185%	16.5s	1067	83.87%	277%
F6	1.0s	27	1.0s	27	100%	1.0s	55	3.03%	100%
F7	5.6s	353	3.7s	212	150%	23.0s	187	34.00%	188%
None	2.9d	134Mio.	29min	111,604	14,126%	7min	26,968	99.39%	58,397%

Table 7.1. The table shows the required number of test cases to detect each failure in the model shown in Figure 7.4, and the time required to execute them. Note that *F3* and *F4* are missing because they were excluded or not detected, respectively, as described. For the heuristic-enabled approach, the accuracy is shown, i.e., what percentage of the heuristic's suggested test cases are below, but next to the boundary of the SOuT. Speedup refers to the required time compared to the time request by depth-first search.

we detected in the SO mechanisms which can be revealed with quite a lot different combinations of component faults and thus identified very early on. However, for *F5* the potential of the approach especially for a bit more sophisticated faults is shown. A possible improvement could be achieved by incorporating knowledge from the test engineer that goes beyond the heuristics shown here. For instance, the test engineer might define several situations to be tested, that might guide the search better than the given heuristics.

7.4.2 Load-Balancing Web-Service—Adaptive Test Case Execution

The SuT for this evaluation was implementation as described in Chapter 6. Within this implementation and MBT approach presented in Chapter 6, we were able to reveal failures, that have been caused by errors made during the implementation of the case study. These failures have been made activatable and deactivatable for the evaluation. The failures that have been detected in the evaluation of MBT are the following (in brackets the phase of the SO mechanism is denoted):

F1: (Distribution) The wrong server was activated resp. deactivated by the proxy. This failure was due to wrong indexing.

F2: (Computation) The proxy did not consider server in the "wrong mode", e.g., a server which can deliver multimedia but is currently in text mode. The proxy did not check whether it was able to switch the mode.

F3: (Distribution) There was a delay of the server activation and deactivation, since the servers waited till the next time step after set to activate, this was due to a wrong working-off order.

F4: (Computation) Active servers were not considered as part of the server pool and thus the decrease was too large.

F5: (Computation) Defect servers were wrongly considered for activation.

For this evaluation, the test model created in Chapter 6 was completed by the criteria for their activation as well as by the option for instantiation, which was set to 1. The criteria introduced for the environment faults are summarized as follows:

1. **Environment Fault Category:** A (or a number of) server(s) within the server pool are deactivated.
   Criterion: A high workload and servers similar in type are currently few.
2. **Environment Fault Category:** A (or a number of) server(s) within the server pool cannot switch the content type
   Criterion: Every situation where a content switch would ease the task of the load balancer
3. **Environment Fault Category:** A (or a number of) server(s) within the server pool is (are) no longer able to be activated or deactivated
   Criterion: Every situation with an increased or decreased load that make activation resp. deactivation of servers necessary

	Depth-First			Adaptive Test Cases			
Failure	time	#test cases	unintended termina-tions	time	#test cases	unintended termina-tion	speedup
F1	5.3s	343	12	6.1s	260	5	175 %
F2	10.6s	743	15	9.6s	531	10	171 %
F3	1.0s	26	2	1.8s	26	0	100 %
F4	1.0s	26	1	1.7s	26	0	100 %
F5	5.8s	371	11	6.2s	271	6	173 %
F6	5min	19,242	604	3.8min	14025	281	172 %

Table 7.2. The table shows the required number of test cases to detect each failure listed, and the time required to execute them. Furthermore, the unintended terminations, i.e., situations where no longer a solution was available and the system has to be restarted, are shown. Speedup refers to the required test cases compared to the test cases request by depth-first search.

After amending the test model by the criteria, the test suite was executed. The results are shown in Table 7.2, compared to the first execution in Chapter 6, named as depth-first search (since the test cases were created by going over all possible test cases in a depth-first approach). All of the failures that have been revealed in a test setting without the extension for adaptive automation are detected using the adaptive test execution. The first difference which is shown results in a lower rate of unintended terminations. An unintended termination is when the boundaries of the SO mechanism are overstepped, i.e., a test case that causes a situation where no SO is possible. The second variation is shown in the number of test cases that was needed to reveal the failures, which was lower in the case of adaptive test cases. However, the time for execution was long, caused by the overhead reasoning and the questioning the behavior distance. Furthermore, a new failure was revealed in this evaluation: **F6** is caused by a missing refresh of the current server status before activating servers. This failure was also detected after letting the depth-first configuration run for a longer time then evaluated before. However, this shows the benefit and the need for test case selection for SO mechanisms: The test end criterion is often time-based. Thus, it is necessary to detect as many failures in the shortest possible time.

Besides being more effective, the here evaluated approach for annotating the purpose of a test case presents a good way of incorporating knowledge of a test engineer into the test suite.

Summary and Outlook. In the previous chapter, we have shown how MBT is applied to SO mechanisms. Different characteristics, mainly derived by the ability of the SO mechanisms to make decisions at run time, have made new concepts necessary. The MBT framework was evaluated according to its abilities to reveal failures in different SOuT and proved its abilities. However, the flat-branching state space of the SO mechanisms, resulting from its abilities to adapt at run time, made the testing endeavor time-consuming. Time-consuming concerning the time for revealing a failure. That is problematic especially without having defined a test end criterion besides a time limit. Depth-first search and random testing drove the test selection in the previous chapter. In this chapter, we took up the challenge to systematize the test selection. The key contributions of this chapter are:

1. Defining error-prone areas in the state space of SO mechanisms: the boundaries of SO.
2. Developing a search-based testing approach for test case selection based on the boundaries of SO
3. Making test suites adaptive by introducing the purpose for the test cases.

The evaluation showed that the new concepts could speed up the revealing of different failures in the case studies of this thesis up to factor 500. Besides being more effective, the insides presented in this chapter offer a test end criterion for SO mechanisms: covering the boundaries of SO. The evaluation showed that most of the failures had been revealed here.

Summary. The techniques for MBT enable beside functional testing also to investigate non-functional properties of SO mechanisms. In this chapter, we show how the performance of SO mechanisms is tested in experimental evaluation. For this purpose, we will systematically derive the requirements for testing the performance of SO mechanisms. These are implemented in the MBT approach presented in the previous chapter, by adapting and extending the concepts. We will explore and derive a metric for quantifying the performance of SO mechanisms. This metric is integrated into the MBT approach, enabling automated performance testing. The abilities are demonstrated by evaluating the concepts on the energy grid and the production cell case study. The content and contributions of this chapter are published in [50, 56].

8 Performance Testing for Self-Organization Mechanisms

8.1 Related Work **152**
8.1.1 Metrics for Adaptation Mechanisms 153
8.1.2 Metrics for SO Mechanisms 159
8.2 Requirements for Performance Metrics for SO Mechanisms **159**
8.3 A Distributed Performance Metric for SO Systems **160**
8.3.1 Time Performance of SO Mechanisms 160
8.3.2 Quality Performance of SO Mechanisms 161
8.4 Performance Evaluation Framework **162**
8.4.1 Generating Unbiased Evaluation Runs 163
8.4.2 Modeling the Environment for Evaluating the Performance of SO Mechanisms 164
8.4.3 Integrating the Evaluation Sequence Selection in the Evaluation Framework 165
8.5 Evaluation **166**
8.5.1 Production Cell 166
8.5.2 Energy Grid 170

The performance of software describes its capabilities in its execution. These capabilities might be determined either by a theoretical analysis or by an experimental evaluation. In general, two measures are of interest: the solution quality and the time taken to achieve the solution [97]. Whereas theoretical analysis is based on abstraction, theorems, and proofs to find an asymptotic bound on the dominant operation under a worst-case or average-case mode, experimental evaluation relies on execution, logging, and measuring according to a set of metrics. The knowledge gained from the performance analysis, theoretical as well as experimental, is used for engineering efficient and effective software. The gain for the engineers highly depends on the quality of the analysis. The quality of performance analysis consists of the accuracy of the results according to the evaluated software. That is the accuracy of selecting the asymptotic boundaries for worst-case or average-case analysis in the theoretical analysis and the accuracy of the metrics describing the software as well as the adequacy of the evaluation conditions for the

experimental analysis. However, achieving a high quality for the analysis is a challenging task [97]. Theoretical as well as experimental analysis are foremost challenged when the system under evaluation is indeterministic, highly parallel, interactive, or highly dependent on unforeseeable run time conditions. All these aspects are characteristics of Self-Organizing, Adaptive Systems (SOASs). A SOAS uses its abilities to reconfigure and restructure itself at run time to cope with an ever-changing environment. Self-Organization (SO) mechanisms are used to fulfill this reconfiguration and restructuring task. An important aspect that is exploited by most of the SO mechanisms is that mostly that process can be carried out locally, i.e., in a small part of the overall system. This aspect makes the SO mechanisms scalable and effective which has a significant impact on the overall system performance. However, it is far from obvious how to design and implement the best performing SO mechanism for a particular system, because SO mechanisms have to operate under ever-changing environmental conditions that are partially unpredictable at design time. This fact demands a powerful performance analysis to support this task. Here, we face the challenge by focusing on an experimental analysis, following van Dyke Parunak & Brueckner [119], who argue that there is a need of empirical evaluation of SO mechanisms because the concepts of theoretical analysis are stretched to their limits given that the majority of SOASs are formally undecidable.

8.1 Related Work

We provide an overview of performance metrics for SO mechanisms and evaluate their abilities within the energy grid case study. Several metrics are identified for adaptation (resp. self-adaptation) mechanisms in the literature. And only very few that are focused on SO, as we describe it in Chapter 4.

To put it simply, the performance of an algorithm describes how well or poorly it works. Self-Organization mechanisms operate on the structure or organization of the system. Consequently, their performance is defined by how well they structure or organize the system. In the literature, different metrics are defined that concretize “how well” algorithms work by identifying several fine-grained performance criteria.

The summary of the state of the art, which is provided next, aims at identifying and discussing different performance metrics concerning their applicability to SO mechanisms. We base our discussion on empirical data we obtained during an evaluation of *PSOPP*’s performance utilizing these metrics. The *PSOPP* algorithm is part of an SO mechanism from the energy grid case study, described in Chapter 2. To apply and evaluate the metrics introduced next, we use the same testbed as the framework described in Section 8.4. It is structured into three main components that encompass the generation of *evaluation suites*, the execution of the evaluation suites, and the observation as well as evaluation of the SO mechanism that is plugged into the evaluation system via an interface.

In our evaluation setting “PSOPP HP”, *PSOPP*’s goal was to create a homogeneous partitioning in each separate subsystem. A homogeneous partitioning is a partitioning in which each partition, i.e., agent group, has a similar average state value. This partitioning is accomplished by minimizing the standard deviation of the average state values by

creating new or dissolving existing partitions and exchanging agents between them. Such a structure has shown to be rather robust against changing states compared to heterogeneous partitions as they are formed in the evaluation setting "PSOPP k-means".

This testbed is used to assess the capabilities of the metrics discussed next. For comparison, we executed all 100 generated evaluation suites, each comprising ten evaluation sequences, in three different settings:

- In the setting "noSO HP", PSOPP was disabled.
- In the setting "PSOPP HP", PSOPP established partitionings according to the homogeneous partitioning objective described above.
- In the setting "PSOPP k-means", PSOPP established heterogeneous partitionings according to the well-known k-means objective function.

All evaluation sequences represented 300 time steps and had been performed in a distributed cluster of 12 computers with an Intel Core-i5 CPU and 4GB RAM for about a week. We performed each setting on a predefined system structure consisting of 1, 2, and 5 separate subsystems.

8.1.1 Metrics for Adaptation Mechanisms

There are several metrics for adaptation (resp. self-adaptation) mechanisms in the literature. As is the case with classical algorithms, they can be clustered into *time-oriented metrics* and *solution-quality-oriented metrics*. The research survey of Villegas et al. [165] as well as the criteria for the evaluation of self-* systems of Kaddoum et al. [84][1] are *time-oriented metrics* that reflect the relationship between time for adaptation and working time. The performance metrics of Becker et al. [15], Tarnu and Tiemann [156], and Reinecke et al. [135] address the *solution quality* of the adaptation mechanism.

Investigated Metrics

Overall, we selected those metrics that apply to SO algorithms, which represent some special form of adaptation. We discuss if the metrics are suitable for measuring the performance of SO mechanisms by our evaluation results.

Time-oriented Metrics Kaddoum et al. [84] extend classical performance metrics to metrics for self-adaptive systems by distinguishing *nominal* and *self-* situations* and focusing on their relation. One example is the WAT metric that is defined as follows:

$$WAT := \frac{\textit{working time}}{\textit{adaptivity time}} \tag{8.1}$$

The codomain of the WAT is $[0, \infty]$, where the performance of the adaptation algorithm is said to increase with the value of WAT. The intuition of WAT is that adaptation

[1] Parts of the criteria for the assessment of adaptive systems have been applied by Cámara et al. [27].

is responsible for keeping the controlled system working with as little disruption as possible by an adaptation mechanism. Further metrics introduced in [84] are also defined as the ratio between adaptation and working time but focus on service-oriented systems, e.g., the response time of a service.

The metrics proposed by Villegas et al. [165] also focus on service-oriented adaptive systems. Proposed information of interest is, for example, monetary execution costs or the reliability of a service according to task completion. Villegas et al. defined the availability (A) resp. unavailability (U) metrics as follows:

$$A := \frac{MTTF}{MTTF + MTTR} \tag{8.2}$$

$$U := \frac{MTTR}{MTTF + MTTR} \tag{8.3}$$

$MTTR$ is the mean time to recover and $MTTF$ is the mean time to fail with a codomain of A and U of $[0, 1]$.[2] A large value of A and a small value of U is desired to attest to an algorithm's good performance. The metrics are based on the concepts of reliability engineering (cf. [93]) and define the performance of an adaptation mechanism over the reliability it yields for the controlled system.

Solution-quality-oriented Metrics

Taranu and Tiemann [156] use a cost function to evaluate the solution quality of an adaptation algorithm in the context of network scenarios. The function maps a performance value to different situations. Each situation may consist of sub-situations with individual costs. Costs are, for example, defined by the generated network traffic, where as little traffic as possible is desired. With regard to a specific situation sit, their performance metric is defined on the basis of the measured costs C_{subsit} and the maximum costs C_{max} of the sub-situations $subsit$:

$$perf(sit) := 1 - \frac{\sum_{subsit \in sit} C_{subsit}}{\sum_{subsit \in sit} C_{max}} \tag{8.4}$$

The metric $perf(sit)$ yields a normalized[3] cost value—resp. a solution quality for a situation sit—on the basis of the costs of different sub-situations, e.g., different time steps. The resulting performance is within the codomain of $[0, 1]$. The best performance is 1.

Becker et al. [15] derive performance metrics from requirements by measuring the time the requirements are fulfilled and, above that, how well they are met. This approach is grounded on requirements specified as *RELAXed* requirement (cf. [36]) and a function that maps the satisfaction resp. dissatisfaction with the requirement within a given time

[2] Note that $A + U = 1$.

[3] In this thesis, the term *normalization* is used in the sense of adjusting values measured on different scales to a notionally common scale of $[0, 1]$.

interval to a numeric value. Let us consider an example where the requirement RF is defined as follows:

"The system SHALL keep the rental fee AS CLOSE AS POSSIBLE to 0."

The function $\Delta(\phi_{RF}, [i, j))$ evaluates the dissatisfaction of the requirement RF in the time interval $[i, j)$, e.g., if there is no rental fee, the value is 0. The corresponding performance metric for a requirement RF is defined as follows:

$$m_{RF} := \begin{cases} 0 & \text{if } \Delta(\phi_{RF}, [i, j)) \geq RF_{max} \\ 1 - \frac{\Delta(\phi_{RF}, [i,j))}{RF_{max}} & \text{else} \end{cases} \tag{8.5}$$

RF_{max} is the rental fee threshold. For m_{RF}, the codomain is $[0, 1]$ with the optimal performance being 1 since the requirement RF is completely fulfilled. The metric is quite similar to Equation (8.4) apart from the property that RF_{max} bounds the solution quality.

Reinecke et al. [135] propose to measure the performance according to the sum of benefits obtained by the decisions made of the adaptation mechanism. To this end, they use three different sets to remember the time steps $i \in \{1, 2, 3, \ldots, N\}$ in which the payoff p_i decreased, did not change, or increased from one time step to another:

$$D_{\ominus} := \{i = 2, 3, \ldots, N \mid p_{i-1} > p_i\}$$
$$D_{\odot} := \{i = 2, 3, \ldots, N \mid p_{i-1} = p_i\}$$
$$D_{\oplus} := \{i = 2, 3, \ldots, N \mid p_{i-1} < p_i\}$$

You can think of these sets as sets of negative, neutral, and positive decisions. The performance metric over these decisions is defined by the following formula that reflects the total benefit of adaptation:

$$Ad := \frac{\sum_{i \in D_{\oplus}} \Delta_i + \sum_{i \in D_{\odot}} p_i}{N - 1} \tag{8.6}$$

Δ_i defines the benefit of a decision as $\Delta_i := \frac{p_i + p_{i-1}}{2}$ and p_i is the benefit in a time step i. Both the function Δ_i and the exclusion of the set $D_{\ominus}$ smooths the payoff function of the system over the considered time. Since the payoff is within $[0, 1]$, the maximum payoff is 1. Ad is normalized by dividing by $N - 1$.

Discussion of the Abilities of Performance Metrics

All metrics have been used to analyze the *PSOPP* algorithm in different settings. These settings are clustered into the number of separate subsystems. Figure 8.1 shows two separate subsystems s_1 and s_2, each systems is equipped with one instance of the PSOPP SO mechanisms forming the Autonomous Virtual Power Plant (AVPP) groups g_n The results of our evaluation are summarized in Table 8.1. In the following, we discuss their significance for SO mechanisms.

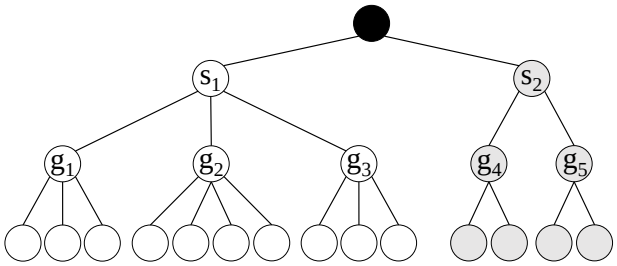


Figure 8.1. The graph shows a possible hierarchy formed by the PSOPP SO mechanism in the energy grid case study. Within this hierarchy, it is possible that just a subgraph, e.g., the right gray part of the graph, is part of a reorganization. The impact on the working time of the entire system is much smaller if the gray subgraph is reorganized than in case of a reorganization of the left white part.

Time-oriented Metrics

The metrics *WAT*, *A*, and *U* (see Equations (8.1) to (8.3)) rely on the ratio between working time and adaptivity time resp. the mean time to fail and the mean time to recover. All three focus on the impact of the adaptation on the working system and reflect the stability as well as the robustness of the organizations established by the SO mechanism. The results of Table 8.1 indicate that *PSOPP HP* is able to achieve more robust partitionings than *PSOPP k-means*. This result reflects our intuition that heterogeneous partitions, as obtained by homogeneous partitioning, are more robust than homogeneous partitions favored by the k-means fitness function (cf. [10]).

Unfortunately, the locality of SO algorithms is neglected by the three metrics. Thus, a reconfiguration in a small part of the system is rated as an adaptation period of the entire system as is for a reconfiguration within a huge part of the system. Although the results in Table 8.1 show that the average number of agents participating in a reconfiguration decreases with an increasing number of subsystems, the whole system is rated "in reconfiguration". This effect contradicts the reality, which is shown in the number of reorganized separated subsystems per reorganization. Ergo, it is, for instance, hard to reason (based on the metrics) whether *PSOPP HP* performs better in a system with 2 or 5 separate subsystems: While *WAT* states that 2 separate subsystems are better, the *A* and *U* metrics prefer 5 separate subsystems.

A drawback of considering only time-oriented metrics is that they do not reflect how well the system performs within an organizational structure. Considering only the time-oriented metrics, it is possible that an SO mechanism that causes the system to work inefficiently is rated very good concerning time if it generates a robust structure. Such a metric is not sufficient to rate the performance of an SO mechanism with all its responsibilities. Therefore, we claim that there is a need for combining time-oriented metrics with solution-quality-oriented metrics to rate the overall performance of an SO mechanism. This claim is also indicated in the evaluation results of *PSOPP k-means* since the high number of reorganizations results in a very high payoff concerning solution quality. An open question is how to aggregate the results of different metrics, e.g., is an

Setting	noSO HP			PSOPP HP			PSOPP k-means		
#Separate Subsystems	1	2	5	1	2	5	1	2	5
WAT	—	—	—	6.92 (1.35)	3.24 (0.90)	0.97 (0.22)	0.02 (0.01)	0.01 (0.01)	0.01 (0.01)
Working Time [s]	—	—	—	29.78 (37.29)	29.69 (92.45)	29.27 (0.21)	15.85 (1.69)	11.57 (3.84)	7.77 (4.84)
Adaptivity Time [s]	—	—	—	4.59 (1.39)	9.93 (2.89)	31.57 (6.62)	623.411 (76.31)	907.80 (151.51)	1,617.07 (394.80)
A	—	—	—	0.77 (0.02)	0.66 (0.02)	0.44 (0.03)	0.02 (0.01)	0.01 (0.01)	0.01 (0.01)
U	—	—	—	0.22 (0.02)	0.33 (0.02)	0.55 (0.03)	0.97 (0.01)	0.98 (0.01)	0.99 (0.02)
$MTTR$ [s]	—	—	—	3.97 (0.19)	5.41 (1.08)	5.43 (1.29)	4.43 (0.09)	5.03 (0.67)	7.32 (0.85)
$MTTF$ [s]	—	—	—	14.13 (1.82)	10.83 (3.18)	4.51 (1.75)	0.11 (0.04)	0.07 (0.05)	0.04 (0.04)
$perf(sit)$	0.52 (0.09)	0.52 (0.07)	0.96 (0.01)	0.96 (0.01)	0.96 (0.01)	0.57 (0.07)	0.99 (0.01)	0.99 (0.01)	0.99 (0.01)
Ad	0.14 (0.03)	0.14 (0.02)	0.01 (0.01)	0.01 (0.01)	0.01 (0.01)	0.14 (0.03)	0.37 (0.09)	0.14 (0.03)	0.04 (0.01)
#Reorganized Separate Subsystems	—	—	—	1.15 (0.37)	2.98 (0.91)	10.21 (2.16)	140.47 (16.93)	245.00 (42.52)	497.63 (123.17)
#Reorganized Separate Subsystems per Reorg.	—	—	—	1.00 (0.00)	1.49 (0.50)	1.64 (1.42)	1.00 (0.00)	1.33 (0.47)	2.24 (1.19)
#Reorganized Agents per Reorg.	—	—	—	1000.00 (0.00)	646.78 (385.14)	283.87 (318.04)	1000.00 (0.00)	714.74 (305.04)	479.35 (284.08)

Table 8.1. Evaluation results for the three settings "noSO HP", "PSOPP HP", and "PSOPP k-means" with different numbers of separate subsystems. All values are averages over 1000 evaluation sequences; values in parenthesis denote standard deviations.

organizational structure's robustness better than one that enables the system to work efficiently?

Solution-quality-oriented Metrics

To rate the performance of an SO algorithm, the optimality of its solution plays a crucial role. In the sense of SO, the optimality depends on the quality of the selected organizational structure from the set of all possible structures concerning a given fitness function.

Let us consider the metrics in Equations (8.4) and (8.5) first since both are quite similar in how they measure the normalized fitness of the SO mechanism over time. Note, however, that Equation (8.5) bounds the optimum by RF_{max}. Because of their similarity, we only evaluated the $perf(sit)$ metric.

Challenges that arise during the evaluation of SO mechanisms with the metrics defined in Equations (8.4) and (8.5) are mainly caused by the locality of the SO mechanisms. This characteristic is a significant difference to the adaptation mechanism considered by Becker et al. [15] as well as Taranu and Tiemann [156] who regard a central approach of only one adaptation mechanism within the entire system. In case of multiple subsystems, as is the fact with our evaluation scenarios, the metrics could be applied to the separate subsystems, but it is not obvious how to calculate the performance for the overall system.

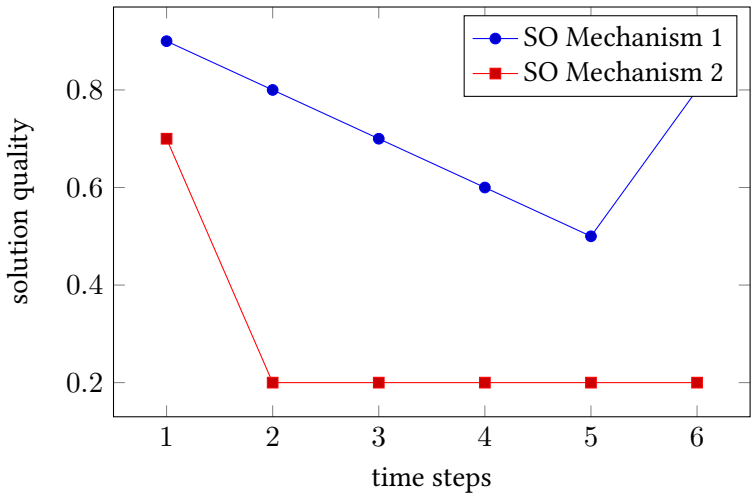


Figure 8.2. The figure shows the payoff of two different series of fitness values (SO Mechanism 1 and SO Mechanism 2) for six time steps. If we apply the metric Ad defined in Equation (8.6) to both series, we get $Ad_1 = 0.2$ and $Ad_2 = 0.1625$, which is not reasonable since the payoff of the *dashed* series is at every time step clearly above the one of the *solid* series.

We used the average value of the $perf(sit)$ metric applied to each separate subsystem as an approximation of the quality of the entire system. However, it is unclear whether this approach reflects the performance adequately. Depending on the intended purpose of the performance measurement, this average value might be weakly informative, e.g., one might want to have the worst rating of all subsystems in case of a risk-averse evaluation.

As shown in Table 8.1, the $perf(sit)$ metric obtains high values in the settings *PSOPP HP* and *PSOPP k-means*, indicating that *PSOPP* performs well in choosing organizational structures. Furthermore, it is possible to compare the performance of an organizational structure without SO (see *no-SO HP* in Table 8.1), i.e., no restructuring or reorganization of the system at run time, with a system with SO: The results indicate that SO with the *PSOPP* algorithm is worthwhile since the fitness value of the system is almost twice as high compared to a system without SO (note that *PSOPP*'s quality was rated using the objective function of homogeneous partitioning in case of *no-SO HP* as well as *PSOPP HP*).

The Ad metric in Equation (8.6) intends to smooth the development of the fitness value used by $perf(sit)$. Alas, the metric shows some adverse side effects that are illustrated in Figure 8.2 where a series of fitness values that is worse than another is rated better. Due to this unwanted effect of the metric, it is hard to use the value for performance evaluation. This is also reflected in the evaluation results of Table 8.1, where the performance of *noSO HP* is rated ten times better than *PSOPP HP*, which is quite incomprehensible considering that Ad only tries to smooth the development of the fitness value that is used by $perf(sit)$.

8.1.2 Metrics for SO Mechanisms

Hitherto, there has not been any work focusing on the design of general metrics for measuring the performance of SO mechanisms. The work of Kaddoum et al. [84] lists metrics under the umbrella of self-* systems and consequently includes self-organization, but the shown metrics, as well as the paper and pencil evaluation, are only considering self-adaptive systems. That is why we introduced their metrics already in the previous section as metrics for self-adaptation mechanisms. Nevertheless, the implementation and design of SO mechanisms has brought along specialized evaluations of the developed algorithms during the recent years (cf. [3, 10, 122]).

Discussion

The development of SO mechanisms includes several specialized evaluations of the performance of the different SO mechanisms, but only in the context of the newly introduced mechanisms (cf. [3, 10, 122]). In general, the evaluation results are used to show the strengths and limitations of the SO mechanisms in a specific setting. Sometimes, the results are also used to optimize the parametrization of the developed mechanisms. Nevertheless, to my knowledge, no general approach addresses systematic and comparable performance evaluations.

8.2 Requirements for Performance Metrics for SO Mechanisms

To realize such an approach for the evaluation of the performance of SO mechanisms, we derive requirements from the results discussed in the previous section and experiences gained during the evaluations of our developed SO mechanisms. The requirements are split into requirements for the metrics themselves and their implementation in a framework for SO mechanism evaluation.

We claim that the following requirements are the most important to be met by metrics to assess the performance of SO mechanisms concerning *time* and *solution quality*:

Req. 1 The *locality* of SO mechanisms has to be taken into account and the aspects of *time* and *solution quality* have to be evaluated within the existing (over run time changing) subsystems that are differently affected by the SO mechanisms, e.g., one subsystem can be reorganized while another keeps on working. Furthermore, it is important to be able to assess the performance of the entire system based on the performance of the subsystems.

Req. 2 Since SO mechanisms have control over the system's structure, their performance strongly influences those of the entire system. So the overhead of a reorganization can be worthwhile if it sufficiently improves the behavior of the controlled system. Consequently, a metric has to take the *benefit of the reorganization* into account.

Req. 3 The interpretation of a value provided by a metric strongly depends on the current state of the system. In self-organizing systems, the possible values for the solution quality can change over time. For instance, a solution quality of 0.7 would be optimal if possible values were defined by the interval $[0, 0.7]$ but quite bad if they stem

from the range $[0, 200]$; the same applies to the parameter *time*. Consequently, there is a need for *dynamic boundaries for the evaluation*—a requirement resulting from the ever-changing environment of SO mechanisms.

To achieve a systematic approach, we claim that there is a need for a framework for performance evaluation that has to satisfy at least the following requirements:

Req. 4 The overall process has to be supported by a *framework for performance evaluation* that is able to systematically evaluate SO mechanisms. The framework's components should support the generation of evaluation runs to perform, the simulation itself, and the application of performance metrics.

Req. 5 To achieve significant results, the evaluation must comprise *simulation runs* that induce an environmental behavior *reflecting probable conditions under which the SO mechanisms have to operate*.

8.3 A Distributed Performance Metric for SO Systems

We use the requirements from the previous section to form a metric that can cope with decentralized SO mechanisms, is defined locally, respects the benefit of a reconfiguration, and handles dynamic boundaries. The performance of a system is composed of two parts: time performance and quality performance. Thus the performance p of a system sys is defined by the following metric:

$$p(sys) = w_t \cdot tp(sys) + w_q \cdot qp(sys), \tag{8.7}$$

where $w_t + w_q = 1$ has to be fulfilled. The factors w_t and w_q enable to weight the importance of the time performance $tp(sys)$ and the quality performance $qp(sys)$. The co-domain of the metric is $[0, 1]$, where a higher value means a better performance of the SO mechanism. This is due to the fact that a bigger value of $tp(sys)$ (cf. Equation (8.10)) means a better ratio of working time compared to reorganization time and a bigger value of $qp(sys)$ (cf. Equation (8.13)) means a better quality achieved compared to the optimal quality. The system sys consists of agents (resp. components) $a \in sys$ which are controlled by the SO mechanism that is analyzed in $r \in R$ evaluation runs accomplished.

8.3.1 Time Performance of SO Mechanisms

Evaluating the time performance $tp(sys)$ requires a clear definition of what the time performance of an SO mechanism is. For classical analysis of time performance the answer is: how long must I wait for my output to appear? [97] Applying that approach to SO mechanisms is not sufficient since we have to deal with two aspects (1) SO algorithms are mostly anytime algorithms that are terminated after a specific time and (2) the quality of the new configuration concerning the time to a next reconfiguration is essential. The goal of the SO mechanism is to supply a configuration for the controlled system that enables it to perform best under ever-changing conditions. The time between the reconfigurations is of interest, too. Since during SO, the controlled system is not able to reach a good performance due to a lousy system structure, or it has even to be stopped in

a safe mode. This effect is an implicit form of the time performance of SO mechanisms. Consequently, we are not only interested in the time to a solution, we are interested in its time impact, i.e., the time used for SO compared to the time where the system runs without disturbances. Due to the characteristics of SO mechanisms, there is no single point where the time could be measured because SO mechanisms solve problems in a distributed fashion. Thus, the calculation is for instance achieved by building a coalition of components that are capable of solving the problem without the rest of the system. This is the case in the production cell case study where a group of robots can find a new configuration if a capability, e.g., a drill, is broken. The measurement is consequently no central affair. The time of reconfiguration, i.e., the time involved in finding a new system configuration needs to be measured for each agent $a \in sys$ as follows:

$$tp(a, r) = 1 - \frac{\sum_{s \in r} \text{reconfigurationTime}(r, a, s)}{\text{duration}(r)}, \tag{8.8}$$

where the time performance is measured by calculating the ratio of the time needed for reconfiguration for a single agent a in a run r consisting of steps s, given by $\text{reconfigurationTime}(r, a, s)$ in the unit time, to the duration of the run r where the measurement has taken place, supplied by $\text{duration}(r)$ in the same unit of time. This calculation of the performance is different in two ways from evaluating traditional mechanisms or algorithms, like in [97]: the measurement is a time ratio and is measured locally. The metric value of Equation (8.8) is prorated with the resulting values of all agents $a \in sys$ to gain the time performance of the system by computing the average of all values:

$$tp(r, sys) = avg_a^{sys} tp(a, r) \tag{8.9}$$

The concrete average function influences the result and has to be chosen with care. The same average function is applied to compute the time performance of the system $tp(sys)$ by prorating the results of all evaluation runs $r \in R$ as follows:

$$tp(sys) = avg_r^R tp(r, sys) \tag{8.10}$$

The codomain of $tp(sys)$ as well as $tp(r, sys)$ and $tp(a, r)$ is $[0, 1]$, and a value close to 1 indicates a better achieved time performance.

8.3.2 Quality Performance of SO Mechanisms

The quality performance $qp(sys)$ determines how good the particular solutions of an SO mechanism have been. However, judging the quality of a solution is highly dependent on the particular SO mechanism as well as the system and its environment. This dependency is because the quality is measured according to the influence of the SO mechanism on the controlled system. The SO mechanism in the production cell case study controls which robot and which cart has to carry out which task. It consequently influences the ability of the system to produce workpieces and effects the overall output resp. throughput. The quality of the SO mechanism is determined by the throughput, the number of processing actions, that the system can apply within an evaluation run.

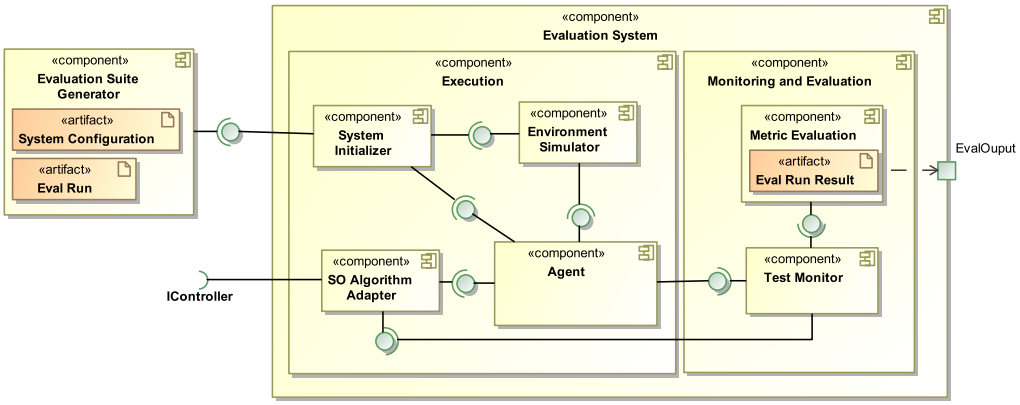


Figure 8.3. The *UML* component diagram shows the essential components of the evaluation framework, consisting of two main components.

This value can be measured locally for each robot, that is processing a workpiece, and then aggregated for the entire system. The SO mechanism in the energy grid case study clusters resp. anti-clusters the power plants into virtual power plants. The optimality of this decision could be measured by the mix-value of the virtual power plants. This value is regio-central, it is determined in the region of every AVPP and has to be aggregated (central) to the overall quality value. The solution quality is evaluated by measuring the domain-dependent quality function quality(r, a) for an evaluation run r of each agent $a \in sys$. As the measured value is dependent upon the context it is normalized by the value which is the best possible one:

$$qp(r, a) = \frac{\text{quality}(r, a)}{\text{quality}_{\max}(r, a)} \tag{8.11}$$

To prorate $qp(r, a)$ for all $a \in sys$ the average is built according to

$$qp(r, sys) = avg_a^{sys} qp(r, a) \tag{8.12}$$

The codomain of $qp(r, a)$ as well as $qp(r, sys)$ is $[0, 1]$, and a value close to 1 indicates a better achieved quality, since 1 would imply that the maximum performance has been reached. To form the proration for $qp(sys)$, different average functions might be appropriate, as described for the time performance before. Thus, $qp(sys)$ is defined as follows:

$$qp(sys) = avg_r^R qp(r, sys) \tag{8.13}$$

8.4 Performance Evaluation Framework

To measure and evaluate the performance of an SO mechanism, it is of utmost importance to establish a testbed where the performance of the mechanism can be analyzed in a systematic, comprehensible, and representative fashion. For the implementation of a framework, we rely on the requirements derived.

We base our evaluation framework shown in Figure 8.3 on the testing framework, introduced in Chapter 5, that has been used for functional testing. Basically, it consists of

an `Evaluation Suite Generator`, an `Execution`, and a `Monitoring and Evaluation` component. The model-based approach is used in all fragments, which is enabled by having executable as well as changeable models. The overall model consists of a static and a dynamic part. The static part is used for enabling automated execution as well as automated evaluation. The dynamic part is used for generating evaluation suites. Based on a model of the configuration the evaluation suite generator derives different possible system configurations for evaluation. The evaluation system is afterward started by initializing the static model that uses `Agents` to set up the environment of the SO mechanism that is plugged into the evaluation framework. The evaluation runs are generated and executed in the same step (we follow an online testing approach for the evaluation of the performance that enables an arbitrary length of evaluation runs). Each phase of the SO mechanism is monitored and evaluated due to the implemented performance metric. The overall evaluation is mainly driven by environmental changes, i.e., changes in the controlled environment of the investigated SO mechanism. In the production cell scenario, such a change might be a faulty robot or drill that causes the SO mechanism to reconfigure the production cell. For the energy grid scenario, a change can be a fluctuation in the production of a power plant due to a changing weather condition of the solar plant leading to a difference in the fulfillment degree of the previously calculated schedule that in turn causes the SO mechanism to reconfigure the AVPP.

8.4.1 Generating Unbiased Evaluation Runs

To supply significant results (fulfilling *Req. 5*), the evaluation runs need to be comprehensible and representative. To illustrate this challenge of signification, let's consider the following example in the production cell scenario: There is only one robot actively using a drill; if the environmental change is to damage the drill, that causes a reconfiguration. However, if the time performance is computed based on the time passed since the last reconfiguration, the environmental action has a direct influence on the performance result of the investigated SO mechanisms. The correlation is unintended since the evaluation should not influence the outcome, the system under evaluation should be the only influence factor. However, it is not possible to completely diminish this influencing factor within a simulation. To achieve the best results, there are two solutions: (1) to evaluate the system with every possible combination of environmental conditions or (2) to select the most representative combination of environmental conditions which represent the reality best. Due to the complexity of the environment, it is not possible to follow option (1).

There is a necessity to select a representative combination of environmental conditions, i.e., take samples for all possible combinations. The result from that sample is a set of evaluation runs. For these evaluation runs we compute a metric, based on Equation (8.7). If this metric value is equal to the metric value for option (1), it is called *unbiased*. An unbiased metric value with a low number of samples is called *efficient* [113]. Sampling could be performed in different manners with different effects. There is the possibility to select random samples from the abstract environment states. Random sampling implies that each possible environmental change has the same probability of being chosen at any

stage during the sampling process to become part of the evaluation run. However, that does not fully reflect the reality of the environment that is modeled and from which the states are sampled. This fuzziness is the case for the production cell as well as the energy grid scenario: The failure rates of the production tools, on the one site, and the weather conditions for the power plants, on the other side, are nonuniform in their occurrences if real-world processes are observed. There are probability distributions that describe the failure rates, that are well investigated in the field of reliability engineering [107], namely mean-time-to-failure rates as well as mean-time-to-repair rates.

8.4.2 Modeling the Environment for Evaluating the Performance of SO Mechanisms

The here presented approach uses a model representation of the environment, where we abstract from concrete states to classes of states. We use a Markov chain model describing how likely it is that a sequence of states occurs or that a state occurs after a particular state, similar to our model explained in Chapter 6, where we formed so-called Environment Profiles (EPs). Here the model is a transition system formed by a pair $(S, \rightarrow)$ where S is a set of states and $\rightarrow$ is a set of transitions from one state to another (i.e., a subset of $S \times S$). That allows to only sample feasible state sequences. Further, the transition system is supplemented by a label $\wedge$ forming a labeled transition system by the tuple $(S, \wedge, \rightarrow)$ where the label annotates the transition probability, forming a Markov chain. That means there are not just the weather conditions *rainy*, *sunny*, and *cloudy*, but there is also the information whether it is possible to have *rainy* weather after *sunny* weather and how probable it is. However, for a sufficiently large number of samples, the random sampling process would still form a comprehensive and representative test sequence. Because the distribution (the statistical variance) of the metric values to the actual value (the value which would be evaluated for all possible situations) is decreasing with the number of samples that would form a fair metric value. That is also known as the law of large numbers. Alas, it is unknown how big large is or should be, and it forms a rather inefficient method. To become more efficient we use the described models, that promise to be a more efficient approach to gain a comprehensive and representative test sequence with the same sample size (cf. [97]). The class of SO mechanism distinguishes the models.

Model for Discrete SO Mechanisms For an SO mechanism with a discrete input space, we apply failure models, from the reliability theory, used for a fault injection approach, as described for the test model in Chapter 6. The faults are injected into the controlled environment of the SO mechanism. That delivers samples as a set of faults (also an empty set is possible) that should be injected, and the latter provides a set of environmental changes.

Model for Continuous SO Mechanisms For an SO mechanism with a continuous input space, we apply the approach of EPs. The resulting evaluation runs are used as repre-

sentative combinations of environmental conditions which represent the reality best and address *Req. 5*.

8.4.3 Integrating the Evaluation Sequence Selection in the Evaluation Framework

In the evaluation framework (cf. Figure 8.3) these models are used to feed the `Environment Simulator` by generating `Eval Runs` in the `Evaluation Suite Generator`. That is done by using S# to execute models, in a way that depending on the mechanism under analysis the corresponding model is executed. The execution has a direct influence on the `Agents` controlled by the mechanism under evaluation. This online evaluation run generating process allows for endless execution runs.

Computing the Length of an Evaluation Run To select the number of all evaluation steps in all runs to be executed, we use the following formula that allows selection of the length by defining the acceptable estimation error $\Delta\mu$ [113]: Therefore we use the length of the confidence interval by Equation (8.14)

$$2z\frac{\sigma}{\sqrt{n}} \tag{8.14}$$

that should be less or equal to the acceptable estimation error (cf. Equation (8.15)).

$$\Delta\mu \geq z\frac{\sigma}{\sqrt{n}} \tag{8.15}$$

By solving Equation (8.15) for n we gain the minimal length of the sequences needed in Equation (8.16).

$$n \geq z^2\frac{\sigma^2}{(\Delta\mu)^2}, \tag{8.16}$$

where n is the number of all evaluation steps in all runs, z is the standard normal distribution (SND) value (taken from an SND table) of the expected distribution, and σ^2 is the standard deviation (SD) to be estimated by taking n evaluation runs. For our evaluation the acceptable estimation error might be $\pm 1\%$ with a confidence interval of 95%. The confidence interval states the probability that the expected metric value $p(sys)$ is within a given symmetric interval of $[p(sys) - b; p(sys) + b]$ where b is called the confidence border. The actual selection of the value

$$b := z\sigma_{p(sys)} \equiv z\frac{\sigma}{\sqrt{n}}$$

is defined by the expected SD and z as the corresponding value of the SND that is expected. Indeed, the number n is the actual number we would like to know and is determined by Equation (8.16) given the confidence interval and the acceptable estimation error. However, the value σ^2 is unknown, since it is the also unknown SD to be estimated by the evaluation runs. Thus, we have to use a rather gross estimate of σ^2. In order to play it safe, σ^2 should be set to 0.25, leading to a rather to big estimate for the value n [113]. So, we select $\sigma^2 = 0.25$ and have the following equation to be

solved having given $z = 1.96$ by taking the value from the SND table and the confidence interval of 0.95:

$$n \geq z^2 \frac{\sigma^2}{(\Delta\mu)^2} = 1.96^2 \frac{0.25}{0.01^2} = 9604$$

8.5 Evaluation

We used two different self-organizing systems for evaluating the accomplishments of our metric described in Section 8.3 and the evaluation framework specified in Section 8.4. Within this evaluation we answer the following research questions:

RQ 1: Is it possible to determine differences in the SO mechanisms' performance during the assessment of the performance of the entire SOAS?

RQ 2: SO mechanisms have a significant impact on the overall performance of the controlled system. Is it possible to quantify this impact reasonably for SO mechanisms?

RQ 3: SO systems are faced with an ever-changing environment; their performance depends on the current run time setting of the system. Are these dynamic performance boundaries reflected in the performance evaluation in a way to provide a comprehensive analysis?

RQ 4: The simulation environment might influence the outcome. Is the evaluation framework able to establish conditions for continuous and discrete SO mechanisms that can produce comparable results in different settings?

These research questions are derived from a set of requirements for metrics and performance evaluation of SO mechanisms, that have been introduced in Section 8.2. We selected the case studies as they represent the two different input spaces of SO mechanisms, described in Chapter 2. Thus, we can demonstrate the two different possible instance of the evaluation framework. To investigate the metric in depth for each case study, we used several different SO mechanisms.

8.5.1 Production Cell

In the production cell case study, we compare a central SO mechanism working with global knowledge with a coalition-formation mechanism with local knowledge only. The centralized mechanism always stops the entire system when a configuration deficiency is detected. It removes the current configuration entirely, computes a new configuration and distributes it. The localized mechanism, on the other hand, forms a coalition of agents, starting with the agent that detected the problem. It recruits more and more neighboring agents until the agents in the coalition can solve the problem at hand among themselves. Only the configurations of those agents within the coalition that must necessarily change their roles is updated. Both mechanisms employ the same algorithm to find a solution within their set of available agents. They differ in the selection of those agents as well as in the method of distribution for the computed solution.

Evaluation Setting

We evaluate both mechanisms within three different setups of the production cell case study: firstly, a setup with few agents (6 robots, 4 carts), and high redundancy with regards to available capabilities (each robot has $\approx 66.7\%$ of the existing capabilities), we refer to this setup as *FA/HR* below; secondly, a setup with more agents (10 robots, 4 carts), and low redundancy (40%) referred to as *MA/LR*; and lastly one with more agents (10 robots, 4 carts), and high redundancy (70%), *MA/HR*. Each model is simulated with both algorithms in several simulation runs. Within each run, environment faults are activated and deactivated randomly according to their respective MTTF and MTTR. For greater comparability, we simulate the systems with the same random seeds, i.e., expose both algorithms to the same environmental conditions.

The numbers of the evaluation are shown in Table 8.2. As the shown data is aggregated over 100 runs with 1000 steps each, the displayed numbers are showing the average value. As outlined in Section 8.4, the concrete choose of the average function is up to the test engineer. One way of choosing is to get more insights into the data. The data is, for this purpose, first tested on a normal distribution. The Shapiro-Wilk test, performed on the data, showed a p-value of < 0.05 which is indicating a nonparametric data set for the measurements. For nonparametric data, it is hard to select an excellent fitting average function. For the given data set the arithmetic mean was evaluated as well as the median. The arithmetic mean is selected in the case when additions of the values are meaningful, while the median is more focused on clustering the data in two equal sized data heaps. That makes the arithmetic more prone to outliers compared to the mean value. Thus, the expectation would be, that the values differ, as we have no normal distribution. However, both values are almost equal for all the data. Leading to the assumption that the data is tightly clustered, which is the case. The data as we process it in the metrics is well suited to addition, used as an average function. Thus, the following data plots the arithmetic average value as the chosen average function. Indeed, it is of further interest to investigate the dispersion of the data. This investigation gives more insights into the state of the data provided. A measure for dispersion is the range of data given. However, for the arithmetic mean the standard deviation (SD) is the average of choice. Indeed, the SD is prone to a normal distribution. However, the data is tightly clustered around the mean. Thus, we opt for the SD in the statistics, delivering the best insight on the dispersion of the given data set. For the further results, the different outcomes of the various configurations and the different settings have been tested according to the independence of the results. The Mann-Whitney-Wilcoxon test has been performed at the given data set to show the independence because no distribution was assumed. The Mann-Whitney-Wilcoxon test resulted in a p-value of < 0.05. Thus, the following discussion of the value is grounded on the given independence.

Discussion of the Evaluation Results

RQ 1: The results of our evaluation as shown in Table 8.2 clearly indicate that the centralized mechanism yields greater quality, i.e., allows for greater throughput. This result is surprising, as one would expect the locality of the coalition-formation mecha-

Mechanism	Centralized			Coalition-Formation		
Model	FA/HR	MA/LR	MA/HR	FA/HR	MA/LR	MA/HR
$tp(sys)$	0.9728 (3e−3)	0.8579 (4e−4)	0.8596 (1e−2)	0.9963 (6e−4)	1.0000 (3e−5)	1.0000 (2e−6)
$qp(sys)$	0.5909 (0.03)	0.9432 (0.06)	0.7022 (0.04)	0.4987 (0.03)	0.7560 (0.04)	0.5779 (0.04)
$p(sys)$ $w_t = 0.5, w_q = 0.5$	0.7819	0.9005	0.7809	0.7475	0.8780	0.7889
$p(sys)$ $w_t = 0.1, w_q = 0.9$	0.6291	0.9347	0.7180	0.5484	0.7804	0.6200
$p(sys)$ $w_t = 0.9, w_q = 0.1$	0.9347	0.8664	0.8439	0.9466	0.9756	0.9577
#Modified Roles per Reconf.	7.85 (2.65)	16.67 (6.67)	13.73 (5.06)	8.41 (3.83)	11.74 (11.17)	10.71 (7.51)
#Reconf. / # Involved Agents per Reconf.	0.45 (0.12)	0.52 (0.13)	0.46 (0.12)	0.88 (0.16)	0.74 (0.19)	0.76 (0.20)
#Steps between Agent Reconf.	6.87 (6.91)	1.99 (2.41)	3.10 (3.41)	12.84 (21.53)	3.83 (5.10)	6.04 (9.49)

Table 8.2. Evaluation results for the two SO mechanisms "Centralized", and "Coalition-Formation" with different production cell setups. All values are averages over 100 evaluation runs with 1000 steps each; values in parenthesis denote SDs.

nism to yield better results since it allows some parts of the system to be reconfigured while other parts keep working. However, this effect was not pronounced enough in our case study to overcome the negative aspects: the coalition has only a subset of the centralized mechanism's knowledge, and it will always prefer a localized solution, leading to little division of labor. On the other hand, we can see that the coalition-formation mechanism has better time performance, i.e., the relationship between working time and reconfiguration time for individual agents is better. This result shows the benefit of not involving every agent in every reconfiguration. The coalition-formation is much more efficient in this respect, for the smallest model almost twice as efficient as the centralized mechanism. As a result, agents can perform more production steps between two subsequent reconfigurations. In reality, the effect on performance would be even more pronounced because physically stopping production costs more time than it does in our simulation. Similarly, changing role allocations would correspond to physical tool changes, also requiring great amounts of time. For larger models, the coalition mechanism outperforms the centralized mechanism in this respect as well. We measured the time performance locally for each agent, computed the performance for the complete evaluation run, and approximate results for the entire system through a series of runs. The quality is measured similar, since the number of processing steps applied by agents within a run, this information is gathered locally for each agent and then aggregated. Hence, to answer *RQ 1*, two different mechanisms are comparable despite their different views on the overall system (local and central). The metrics deliver a clear indication of the advantages of the different mechanisms.

RQ 2: The solution quality can be weighted higher or lower to consider the SO mechanism's influence on the system performance. This enables us to control the influence of the quality parameter with the time parameter. The quality is best measured

in the MA/LR setting, that indicates, that the quality is depended on the actual setting as well. Nevertheless, it is still possible to quantify a recent difference between the quality of the centralized and the coalition setting. That undermines an effect, that is assumed for decentralized mechanisms: it is often stuck to local optima. This effect is reasonably quantified here.

RQ 3: Our metric can also account for the dynamic performance boundaries of SO (*RQ 3*). Remember that the quality ratings equal the system's actual productivity, measured by production actions, e.g., drill, compared to the maximal possible productivity. For the maximal possible quality as referenced in Equation (8.12), we executed a system run for each setup without any environment faults. For the FA/HR setting of 484 production actions was achieved, 336 production actions for the MA/LR setting, and 343 production actions for the MA/HR setting. One step can encompass at most one production action, hence this value abstracts from the actual time required. Thus, a larger system with low redundancies is prone to be less productive due to long transits. Here, the same initial system configuration is used for calculating the maximum quality as well as for starting the evaluation. That initial configuration encompasses, amongst other things, the initial role allocation (i.e., which robot and which cart is applying which capability) of the system. However, when the system encounters a faulty environment, the maximal throughput in ideal conditions may in many cases be unreachable even with the best SO mechanism, which explains the relatively low scores for both algorithms.

Further, high time performance ratings can be achieved by both algorithms: even though a faulty environment leads to more time spent on reconfigurations, it influences the total simulation time, in the same manner, thus limiting its influence on the quotient. By assigning a lower weight to the quality ratings, we can account for this imbalance to some extent. Similarly, there exists a certain disconnect between the time required for processing steps in our simulations and reality, the former being much lower. This disconnect does not exist for reconfiguration times, and therefore it affects the time performance as defined in Equation (8.8). However, it does so equally for both compared mechanisms.

Hence, while the absolute values in the simulation differ from the realization of a real hardware application scenario the metric has to be taken with a grain of salt for the quantitative comparison, the relation between the two mechanisms remains still the same and thus still allows for fair qualitative comparison. To give a complete picture, we also included the number of discrete steps the system makes between two reconfigurations involving the same agent, on average.

RQ 4: Lastly, we consider the evaluation system's influence on our results in *RQ 4*. The SD for time performance and quality in our evaluation results is low, less than 0.01 for time performance, and less than 0.06 for the quality rating. We can thus assume the results are unbiased. The high SDs for per-reconfiguration results (the last three rows) is expected: the number of necessary reconfiguration changes and the frequency of reconfigurations depends on the respectively occurring environment faults, whose frequency and impact vary greatly.

8.5.2 Energy Grid

For the evaluation within the self-organized creation of virtual power plants in a smart grid we used an SO mechanism called *PSOPP* [10] (*P*article *S*warm *O*ptimizer for the *P*artitioning *P*roblem). The PSOPP is a particle swarm optimizer that partitions a set of agents representing a (sub)system into pairwise disjoint and non-empty groups. These groups constitute the (sub)system's configurational structure. Feasible organizational structures can be described by so-called partitioning constraints that restrict the number and the size of these groups. PSOPP is an anytime algorithm and a metaheuristic that optimizes the groups' composition concerning an objective function. In our evaluation, PSOPP is used to optimize the groups' composition in each so-called separate AVPPs of a hierarchically structured system.

Evaluation Setting

We executed 100 generated evaluation runs, each comprising 300 evaluation steps leading to a size of the evaluation run that is bigger than the smallest usable size calculated using Equation (8.16). To investigate two different SO mechanisms we instantiated the PSOPP algorithm with two different settings: (1) In the setting *PSOPP HP*, PSOPP established partitionings according to an homogeneous partitioning objective function defined in [10]. (2) In the setting *PSOPP k-means*, PSOPP established heterogeneous partitionings according to the well-known k-means objective function. All evaluation runs have been performed in a distributed cluster of 12 computers with an Intel Core-i5 CPU and 4GB RAM for about a week. We performed each setting on a predefined system structure consisting of 1, 2, and 5 separate subsystems and 1000 controlled power plants within the system. This structure is called regio-central since we each subsystem is centrally organized.

The results of our evaluation are summarized in Table 8.3. Having a closer look at Table 8.3 the performance metric $p(sys)$ (cf. Equation (8.7)) is shown for all instances with three different configurations according to their weights, a balanced weighting, a favor for quality, and a favor for the time. The data is computed by using the arithmetic mean value with the according to SD from 100 different evaluation runs. The selection of the average function here followed the same approach as described in the setting of the production cell. The investigated data are also not normal distributed, but tightly clustered, making the mean value a good fit for describing the data. The normal distribution was tested, and the p-value resulted in < 0.05. Further, the independence between the values to compare was tested by the Mann-Whitney-Wilcoxon test with a resulting p-value< 0.05. Overall we have observed very slight variants of the performance over the runs and observed no big outliers.

Discussion of the Evaluation Results

RQ 1: A first observation is the fact that the decrease of agents involved in the reconfiguration has a rather low impact on the $tp(sys)$ value in both types of SO mechanisms. That effect is also shown in the number of reorganizations performed in the different setting compared with the involved number of agents in a reorganization. The more

Setting	PSOPP HP			PSOPP k-means		
#Separate Subsystems	1	2	5	1	2	5
$tp(sys)$	0.87 (0.11)	0.86 (0.27)	0.90 (0.02)	0.02 (0.002)	0.02 (0.004)	0.02 (0.007)
$qp(sys)$	0.96 (0.02)	0.96 (0.01)	0.96 (0.01)	0.99 (0.01)	0.99 (0.01)	0.99 (0.01)
$p(sys)$ $w_t = 0.5, w_q = 0.5$	0.92 (0.07)	0.91 (0.15)	0.94 (0.02)	0.51 (0.03)	0.51 (0.05)	0.51 (0.09)
$p(sys)$ $w_t = 0.1, w_q = 0.9$	0.95 (0.04)	0.95 (0.05)	0.95 (0.01)	0.89 (0.03)	0.89 (0.06)	0.89 (0.10)
$p(sys)$ $w_t = 0.9, w_q = 0.1$	0.88 (0.04)	0.87 (0.19)	0.91 (0.02)	0.12 (0.02)	0.12 (0.03)	0.12 (0.06)
#Reorganized Separate Subsystems	1.05 (0.32)	2.88 (0.88)	11.01 (1.98)	141.57 (17.93)	244.11 (43.11)	501.36 (113.11)
#Reconfigured Agents per Reconf.	1000.00 (0.00)	696.78 (400.14)	252.47 (288.34)	1000.00 (0.00)	734.74 (335.52)	499.51 (284.18)

Table 8.3. Evaluation results for the two settings "PSOPP HP" and "PSOPP k-means" with different numbers of AVPPs. All values are averages over evaluation runs with 300 steps; values in parenthesis denote SD.

separate subsystems, the fewer agents are on average involved in a reorganization, but also the more reorganizations are necessary for keeping up the goals of the Corridor of Correct Behavior (CCB). This effect seems to be an effect of the regio-central knowledge that is lower than the central knowledge and thus leads to a higher need for reconfigurations. This effect is reflected in the metric, by having almost the same value despite a changing subsystem size. All these local effects are handled in the metric.

Same for $qp(sys)$, all values have been gathered locally. The value for the quality function for PSOPP HP setting is stating how similar the AVPPs are in their composition. Thus, the goal it the minimization of the SD of the average state values of the power plants in each AVPPs. For the k-means setting, the similarity of the average state is the measure of quality for each AVPP. For $qp(sys)$, in the HP and the k-means setting, similar effects are shown as for the time performance: the increasing number of separate subsystems has no impact on the quality of the system. Measuring the performance local is consequently able to judge over the global system without neglecting the structure of the system and the SO mechanisms.

RQ 2: Having $p(sys)$ for the two different SO mechanisms in the scenario we can observe that the homogeneous partitioning is in favor. That reflects the fact that homogeneous partitioning is more robust than k-means, as described by Anders et al. [10]. However, the robustness has a slight price in quality, that is overall more optimal with k-means (see the $qp(sys)$ values). Nevertheless, that comes with a high price of a very poor $tp(sys)$ result. To achieve a better rating for PSOPP k-means a possible allocation of the weights is $w_t = 0.03$, $w_q = 0.97$. However, it is not recommended to choose such a strong favor for one part of $p(sys)$ since it ignores one of the two crucial performance factors. Thus, to answer *RQ 2* the benefit can be considered, and even more the influence can be steered individually.

RQ 3: The answer to *RQ 3* is shown in the fact that we observed fluctuations throughout the evaluation runs within the maximum. That is different from the production cell case study, where the maximum for the quality performance was computed for a run, not for a step. In this case study, the maximum value is dynamically calculated at each step for a single subsystem, since the value is depended on the current state of the controlled power plants in a subsystem. That is highly necessary to normalize the different achievements in the various system steps.

RQ 4: *RQ 4* questions whether the results are adequate regarding the conditions under which they were measured. This question is hard to answer with the resulting data since we have no gold standard to compare with. However, our argumentation of Equation (8.12) indicates that we have an accuracy of at least 0.95 for the measurements. The inaccuracy of 0.05 is within the variation of the $p(sys)$ value according to the SD and consequently negligible. Thus, we have established an adequate evaluation framework.

Summary and Outlook. In this chapter we showed how the Model-Based Testing (MBT) approach of this thesis is enabling to judge over the capabilities of SO mechanisms in its execution, i.e., to assess the performance of SO mechanisms. Testing the performance of SO mechanisms is a systematic, experimental evaluation of the performance. We started by carving out the requirements for this endeavor. The characteristics of the SO mechanism determine these. Concerning the dimensions of performance, time and solution quality, SO mechanism differ: The time is split in the time the SO mechanism is consuming for reorganization, but also the time the solution is stable afterward. The interplay between the SO mechanisms and their environment and the controlled system are of importance. We showed how to incorporate the properties of the SO mechanism into a metric. The important aspect here was to gather the information locally at each component of the system that is controlled by the SO mechanisms and within the SO mechanisms themselves. The interplay of SO mechanism and its controlled environment is also challenging for testing the performance under the right conditions. These conditions have been defined by a situation that is unbiased to the result of the evaluation. We discussed these aspects and presented a concept for modeling the environment in a probabilistic way to achieve results that are statistically independent and thus comparable for continuous and discrete SO mechanisms. The evaluation shows how to use these achievements for testing the performance on different SO mechanisms in the energy grid and the production cell case study.

---

9 Conclusion and Outlook

The aim of this thesis was to provide an approach for testing SO mechanisms in an MBT approach. The previous chapters have shown a thorough approach providing automated testing of SO mechanisms. The following summarizes the contributions of the approach as well as the evaluation results that demonstrate the approach's capabilities and applicability to SO mechanisms. By providing the MBT approach for SO mechanisms in this thesis, new research challenges, and future directions have opened that are discussed at the end of this chapter.

9.1 Summary of Research Contributions and Evaluation Results

The provisioning of an MBT approach for SO mechanisms has been achieved by six major contributions that form this thesis. Each contribution has been thoroughly evaluated by the applications of the proposed concepts to five different case studies, which have been introduced in this thesis. The case studies have been chosen from different research communities and institutions to show the generality of the proposed approach.

Establishing a Notion of Failure for Self-Organization Mechanisms

A failure is the deviation of the observed behavior from the expected behavior in a defined situation. This defined situation is in this thesis a test case that provides an input to the Self-Organization Mechanism under Test (SOuT). However, the expected behavior of an SO mechanism is not entirely defined: We do know the correct situations for a SOAS, but we do not know the correct transitions into these situations or states. The correct situations are defined via the CCB, which constraints unintended situations. These unintended situations, however, are (as shown in the Restore Invariant Approach (RIA) [70, 109, 110]) restorable by the SO mechanism. Thus, a violation of the correct behavior is not directly a failure, in contrast to a common failure definition (cf. [108]). For software testing, i.e., for revealing failures by executing the software, a concrete definition of a failure is needed. This definition is provided in this thesis as follows: The CCB defines the intended behavior. The SO mechanism needs to detect every violation, compute a new system configuration for the system (if there is one available) and needs to distribute this configuration. If the SO mechanism does not show this behavior, a failure occurs. This contribution is the fundament for the overall achievement of this thesis: MBT for SO mechanisms. We have shown in different case studies, that it is possible to specify an CCB and use for revealing failures. The revealed failures have shown to be an unintended behavior of the SO mechanism, caused by an error.

Establishing Testability for Self-Organization Mechanism

Having a failure definition is the first, crucial step toward testing software. The next step is to be able to observe the behavior of the tested system and to control it. These two aspects are needed for executing test cases. First, by observing the System under Test (SuT) to perceive its current state to reveal a failure. Second, by controlling the system to establish the test conditions needed and providing the input as specified in the test case. Both properties are summarized as testability. Testability for SO mechanism has been achieved in this thesis with the proposed architectural pattern of the Corridor Enforcing Infrastructure (CEI). The CEI meets the challenges of making SO mechanisms testable. That challenges are since SO mechanisms are highly interwoven with the controlled system and its environment. By applying the CEI pattern, the SO mechanisms are controllable by having defined phases of SO (in a feedback-loop-oriented process) and defined interfaces. These interfaces are also allowing for observability. It is possible to use the CEI for making SO mechanism testable that implement a feedback-loop-oriented SO mechanism and not explicitly the CEI. Most of the engineered SO mechanisms follow this approach [22].

We have been able to show for each of the five case studies (where only two of them explicitly implemented the CEI) that the CEI enabled us to test the SO mechanisms. The CEI pattern has been applied to the generation of the test scaffold for every case study and enabled us to observe and control the SO mechanism.

Enabling to isolate and integrate Self-Organization mechanisms for systematic testing

The CEI further allows for isolation and integration of SO mechanisms. This decomposition and composition of an SuT has proven in theory [169] as well as praxis [17, 121] as the most effective and efficient method for revealing failures. Testing SO mechanisms is no exception, as shown in the evaluation made in this thesis. Isolation and later integration of the components of the SO mechanisms follow the divide and conquer principle of testing and is needed to cope with the challenges that arise from testing SO mechanisms, foremost, coping with error masking and handling the vast, flat branching state space. The presented concepts for isolation and later composition of the SO mechanisms set the stage for different test techniques to act at different test stages. The contribution is set into a test architecture, offering a complete test scaffold for SO mechanisms. This test architecture has been implemented for testing each of the five used case studies in this thesis and proved its success. In this test architecture, staged testing is enabled for SO mechanisms. Here, the environment of the SO mechanism is the decisive integration factor. This way of integration is newly developed for testing SO mechanisms.

Establishing a closed-loop Model-Based Testing approach suited for testing Self-Organization mechanisms

The realization of the test architecture was done in an MBT approach, that is designed and suited for SO mechanisms. The characteristics of SO mechanisms make it necessary to cope with the adaptive behavior at run time. For this purpose, this thesis offers

an extension of the known MBT approach by implementing feedback and using the so-called run time models to reflect the state of the SuT and its environment back into the test model. That enriched test model enabled us to use the test model as a complete test scaffold for an SO mechanism. Further, the models incorporate the ability to derive test cases during execution and apply them directly on the SOuT. The result of a test case is present in the model, due to the closed-loop, and thus the test oracle is working on the test model. The MBT approach proposed in this thesis offers complete test automation for SO mechanisms. It offers the ability to either use a fault-based or a probabilistic model for the two kinds of SO mechanism: the continuous and the discrete one. As the task of engineering the complete test model is complex and demanding (e.g., the used source code of the test model of the production cell case study encompasses over 5,000 SLOC), an Back-to-Back (BtB) testing approach was developed. The BtB testing approach is based on the idea of co-development and is enabled by the scaffolding abilities of the MBT approach for a very early development stage. The result of BtB is a set of tested requirements for the development as well as for testing. The evaluation proved this approach as very useful. Overall, each of the five case studies has been thoroughly tested in the MBT approach. We were able to reveal failures in the different setting. The failures showed a particular pattern: most of them were located at the boundaries of SO, i.e., the point in the state space where rarely a reconfiguration is possible, due to the lack of redundancy.

Establishing a search-based test case generation approach for efficient testing of Self-Organization mechanisms

The observation of the particular characteristics of the failures observed in the evaluation was exploited. The thesis established the term of boundary-interior-testing for SO mechanisms for specifying this characteristic. Boundary-interior-testing for SO mechanisms is a vital test requirement developed in this thesis that is suited for SO mechanisms. A search-based test case generation approach was presented, that generates test cases that are directed towards these boundaries. The search-based approach was suited for this search problem using a breadth-first search method combined with specific heuristics. As shown in the evaluation, the contribution enabled to reveal the same kind of failures up to 500 times faster than an approach without a directed test case generation.

Extending the Model-Based Testing approach for testing the performance of Self-Organization mechanisms

The MBT approach for testing is able to cope with functional testing, but also offers the ability to test non-functional aspects. In this thesis, an approach for measuring and evaluating the performance of SO mechanisms was presented. For this purpose, a definition of performance for SO mechanism was derived in the foundation of the CCB. Thus, the time performance is focused on how long the SO mechanism enables the SOAS to stay inside the corridor and how long the SO mechanism takes time outside the corridor to reconfigure the system. This performance measure is different from

the performance of standard algorithms that are just focused on the time for finding a solution. Besides the time performance, also a notion of quality performance of SO mechanisms was given in the thesis. In order to determine the performance of an SO mechanism the MBT approach was extended by concepts that allow for measuring the performance under unbiased conditions. The evaluation enabled to judge over the performance of the investigated systems, and we were able to show different effects of performance of an SO mechanism. Thus it is possible to answer the following: at what size of the system does a decentralized SO mechanism outperform a central one? Is heterogeneous clustering of organizational units better than homogenous?

9.2 Open Research Challenges and Future Directions

This thesis provides an approach for testing SO mechanisms in an MBT approach. It laid the ground for research of systematic quality assurance of SOAS. In this section, we summarize open research challenges and outline a few directions for future research.

Fault Localization for Self-Organization Mechanisms

During the evaluation of the approach presented in this thesis different failures have been revealed. However, having the information of the deviation between the actual and the intended behavior is not sufficient for removing the error leading to the particular failure. For this purpose, the failure has to be tracked down to the fault, causing the failure, that is due to the error made by a programmer. That knowledge is needed for removing the fault and preventing the failure. As shown by Wong et al. [167], a variety of approaches is available in the literature that addresses this problem of fault localization. However, the application of fault localization to SO mechanisms is far from obvious. As shown in this thesis, SO mechanisms are highly addicted to their underlying state space, their environment. That aspect needs to be respected in fault localization. The given concepts need to be investigated on their abilities to include that aspect and need to be extended. The resulting approach of fault localization for SO mechanism will ease the development of reliable SO mechanisms.

Mutation Operators for Self-Organization Mechanisms

Mutation operators are patterns for source code modifications with the aim of introducing faults. The faults are consequently known and are used for the evaluation of a test approach. In this thesis mutants where extracted from software repositories to evaluate the proposed testing approach. Further, common mutation operators were applied to introduce faults in the code of the SOuTs. Thus, it was able to demonstrate the ability of the approach. However, this customized solution makes it hard to compare the ability of testing approaches for SO mechanisms. Standard mutation operators will enable this comprehensibility for future testing approaches that are also suited for SO mechanisms. Consequently, it is necessary to in-depth investigate the characteristics of SO mechanisms with the aim of finding adequate mutation operators, suited for SO mechanisms. In [134], we already started this investigation which needs to be extended and standardized for the evaluation of the testing approach for SO mechanisms.

A Testbed for Self-Organizing, Adaptive Systems

Having a suited approach for testing SO mechanisms is the first step for thorough quality assurance of SOAS. The SO mechanisms are a curial part of a SOAS, as discussed throughout the thesis. Based on the results it is now possible to extend the SuT and incorporate further parts of the SOAS into testing. For this purpose, a testbed will be needed that can scaffold the SuT. One particular concern here is the distribution of the deployed SOAS. Thus, challenges of synchronization for the evaluation need to be extended, and concepts for controlling the distributed system needs to be made. The first challenge is to create a consistent snapshot of a distributed system, to provide observability. This issue is in parts already addressed in this thesis, but the assumptions made, need to be validated and reconsidered for a complete integration test of a SOAS. Merayo et al. [99] provide an approach for this research direction, where a passive testing concept is developed and integrated with a monitor environment that is focused on communication. This paper might be a promising starting point for the observation of a distributed SOAS. The problem of controllability is neglected by Merayo et al. [99], as they focus on passive testing. Controlling the system to execute test cases in a defined manner (the steps are executed in a defined order) is demanding since most of the test driver used for controlling the SuT vitiate the results by interfering the SuT. These challenges need to be addressed for testing integrated SOAS.

Extending the Concepts of Testing Self-Organization Mechanisms to Machine Learning

An SO mechanism can autonomously make decisions and adapt itself as well as the controlled system at run time. This characteristic leads to underspecification of the SuT, addressed in this thesis. Machine learning mechanisms are also in some sort underspecified, as they are specified resp. trained by examples. The examples are included in a training and validation set and used for learning. Thus, SO mechanisms and machine learning mechanisms have in common that they are underspecified. Machine Learning mechanisms are commonly validated before they are released, this is similar to testing. However, only an accuracy value is provided, a low one indicates a lousy learning result and a high one a good learning result. Failures in the implementation or the learning procedure are not addressed. Further, failure is not defined at all, only if the system falls into an exception or similar. With the rapid speed, machine learning mechanisms are spreading in nowadays software, a more thorough process for testing is needed. As similar concepts and similar challenges arise as for testing SO mechanisms, the contributions of this thesis might be a good starting point toward a quality assurance for machine learning.

Bibliography

[1] IBM ILOG CPLEX Optimization Studio. `https://www.ibm.com/de-de/marketplace/ibm-ilog-cplex`, 2018. Accessed: 2018-06-08.

[2] A. Abdurazik and J. Offutt. Using Coupling-Based Weights for the Class Integration and Test Order Problem. *The Computer Journal*, 52(5):557–570, 2009.

[3] M. Al-Zinati and R. Wenkstern. A Self-Organizing Virtual Environment for Agent-Based Simulations. In *Proc. fo the 14th Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 1031–1039. Int. Foundation for Autonomous Agents and Multiagent Systems, 2015.

[4] R. T. Alexander, J. M. Bieman, S. Ghosh, and B. Ji. Mutation of Java Objects. In *Proc. of the 13th Int. Symp. on Software Reliability Engineering*, pages 341–351. IEEE, 2002.

[5] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An Orchestrated Survey of Methodologies for Automated Software Test Case Generation. *Journal on Systems and Software*, 86(8):1978–2001, 2013.

[6] G. Anders. *Self-Organized Robust Optimization in Open Technical Systems: Self-Organization and Computational Trust for Scalable and Robust Resource Allocation under Uncertainty* . PhD thesis, University of Augsburg, 2017.

[7] G. Anders, H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif. Decentralized Reconfiguration for Self-Organizing Resource-Flow Systems Based on Local Knowledge. In *Proc. of the 8th IEEE Conf. Wsh.s Engineering of Autonomic and Autonomous Systems*, pages 20–31. IEEE, 2011.

[8] G. Anders, F. Siefert, J.-P. Steghöfer, and W. Reif. A Decentralized Multi-agent Algorithm for the Set Partitioning Problem. In I. Rahwan, W. Wobcke, S. Sen, and T. Sugawara, editors, *Proc. of the 15th Int. Conf. on Principles and Practice of Multi-Agent Systems*, volume 7455 of *LNCS*, pages 107–121. Springer, 2012.

[9] G. Anders, F. Siefert, N. Msadek, R. Kiefhaber, O. Kosak, W. Reif, and T. Ungerer. TEMAS – A Trust-Enabling Multi-Agent System for Open Environments. Technical report, University of Augsburg, 2013.

[10] G. Anders, F. Siefert, and W. Reif. A Particle Swarm Optimizer for Solving the Set Partitioning Problem in the Presence of Partitioning Constraints. In *Proc. of the 7th Int. Conf. on Agents and Artificial Intelligence*. SciTePress, 2015.

[11] W. R. Ashby. Principles of the Self-Organizing Dynamic System. *The Journal of General Psychology*, 37(2):125–128, 1947.

[12] U. Aßmann, S. Götz, J.-M. Jézéquel, B. Morin, and M. Trapp. A Reference Architecture and Roadmap for Models@run.time Systems. In N. Bencomo, R. France, B. H. C. Cheng, and U. Aßmann, editors, *Models@run.time: Foundations, Applications, and Roadmaps*, pages 1–18. Springer, 2014.

[13] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. on Software Engineering*, 41(5):507–525, 2015.

[14] T. Bauer and R. Eschbach. Enabling Statistical Testing for Component-Based Systems. In K.-P. Fähnrich and B. Franczyk, editors, *GI Jahrestagung*, volume 176 of *LNI*, pages 357–362. GI, 2010.

[15] M. Becker, M. Luckey, and S. Becker. Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time. In *Proc. of the 9th ACM SigSoft Int. Conf. Quality of Software Architectures*. ACM, 2013.

[16] F. Belli, A. Hollmann, and S. Padberg. Communication Sequence Graphs for Mutation-Oriented Integration Testing. In *Proc. of the 3rd IEEE Int. Conf. on Secure Software Integration and Reliability Improvement*, pages 387–392, 2009.

[17] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[18] R. V. Binder, B. Legeard, and A. Kramer. Model-based Testing: Where Does It Stand? *Communications of the ACM*, 58(2):52–56, 2015.

[19] J. S. Bradbury, J. R. Cordy, and J. Dingel. Mutation Operators for Concurrent Java (J2SE 5.0). In *Proc. of the 2nd Wsh. on Mutation Analysis*. IEEE Computer Society, 2006.

[20] L. C. Briand, Y. Labiche, and Y. Wang. An Investigation of Graph-Based Class Integration Test Order Strategies. *IEEE Trans. on Software Engineering*, 29(7):594–607, 2003.

[21] M. Broy, B. Jonsson, J. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, volume 3472 of *LNCS*, 2005. Springer.

[22] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering Self-Adaptive Systems through Feedback Loops. In B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer, 2009.

[23] F. Buschmann, K. Henney, and D. Schimdt. *Pattern-Oriented Software Architecture: On Patterns and Pattern Language*, volume 5. Wiley, 2007.

[24] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conf. on Computer and Communications Security*, pages 322–335. ACM, 2006.

[25] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive Software Needs Quantitative Verification at Runtime. *Communication of the ACM*, 55(9):69–77, 2012.

[26] J. Cámara and R. de Lemos. Evaluation of Resilience in Self-adaptive Systems using Probabilistic Model-checking. In *Proc. of the 7th Int. Symp. Software Engineering for Adaptive and Self-Managing Systems*, pages 53–62, 2012.

[27] J. Cámara, P. Correia, R. de Lemos, and M. Vieira. Empirical Resilience Evaluation of an Architecture-based Self-adaptive Software System. In *Proc. of the 10th Int. ACM Sigsoft Conf. on Quality of Software Architectures*, pages 63–72. ACM, 2014.

[28] S. Camazine, J.-L. Deneubourg, N. R. Franks, J. Sneyd, E. Bonabeau, and G. Theraula. *Self-organization in Biological Systems*, volume 7. Princeton University Press, 2003.

[29] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems, Second Edition*. Springer, 2008.

[30] J. Chaplin, O. Bakker, L. de Silva, D. Sanderson, E. Kelly, B. Logan, and S. Ratchev. Evolvable Assembly Systems: A Distributed Architecture for Intelligent Manufacturing. 48(3): 2065 – 2070, 2015. Proc. of the 15th IFAC Symp. on Information Control Problems in Manufacturing.

[31] K. Chen, J. Powers, S. Guo, and F. Tian. CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds. *IEEE Trans. on Parallel and Distributed Systems*, 25(6):1403–1412, 2014.

[32] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical report, Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

[33] T. Y. Chen, J. Feng, and T. Tse. Metamorphic Testing of Programs on Partial Differential Equations: A Case Study. In *Proc. of the 26th Annual Int. Computer Software and Applications Conference*, pages 327–333. IEEE, 2002.

[34] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The Case for Evaluating MapReduce Performance Using Workload Suites. In *IEEE 19th Int. Symp. on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 390–399, 2011.

[35] Y. Chen, S. Alspaugh, and R. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. of the 38th Int. Conf. on Very Large Data Bases*, 5(12):1802–1813, 2012.

[36] B. Cheng, P. Sawyer, N. Bencomo, and J. Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*, pages 468–483. Springer, 2009.

[37] S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, 2008.

[38] S.-W. Cheng, D. Garlan, and B. R. Schmerl. Evaluating the Effectiveness of the Rainbow Self-Adaptive System. In *Proc. of the 5th Wsh. on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141. IEEE Computer Society, 2009.

[39] M. Cohn. *Succeeding with Agile: Software Development using Scrum*. Pearson, 2010.

[40] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[41] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Int. Symp. on Operating System Design and Implementation*, pages 137–150, 2004.

[42] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[43] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4):34–41, 1978.

[44] B. Demuth and C. Wilke. Model and Object Verification by Using Dresden OCL. In *Proc. of the Russian-German Wsh. Innovation Information Technologies: Theory and Practice*, pages 81–90, 2009.

[45] B. Eberhardinger. Test Case Generation for Self-Organising Algorithms. In *Organic Computing: Doctoral Dissertation Colloquium*, volume 7. Kassel University Press, 2015.

[46] B. Eberhardinger. Testing Self-Organizing, Adaptive Systems (Best Paper Award). In *Proc. of the 9th IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 140–145. IEEE, 2015.

[47] B. Eberhardinger, J.-P. Steghöfer, F. Nafz, and W. Reif. Model-driven Synthesis of Monitoring Infrastructure for Reliable Adaptive Multi-Agent Systems. In *Proc. of the 24th IEEE Int. Symp. on Software Reliability Engineering*, pages 21–30. IEEE, 2013.

[48] B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif. Towards Testing Self-Organizing, Adaptive Systems. In *Proc. of the 26th IFIP Int. Conf. on Testing Software and Systems*, pages 180–185. Springer, 2014.

[49] B. Eberhardinger, G. Anders, H. Seebach, F. Siefert, and W. Reif. A Framework for Testing Self-Organisation Algorithms. *GI Softwaretechniktrends*, 35(1), 2015.

[50] B. Eberhardinger, G. Anders, H. Seebach, F. Siefert, and W. Reif. A Research Overview and Evaluation of Performance Metrics for Self-Organization Algorithms. In *Proc. of the 9th IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 122–127. IEEE, 2015.

[51] B. Eberhardinger, A. Habermaier, H. Seebach, and W. Reif. Back-to-Back Testing of Self-Organization Mechanisms. In *Proc. of the 27th IFIP Int. Conf. on Testing Software and Systems*, pages 18–35. Springer, 2016.

[52] B. Eberhardinger, G. Anders, H. Seebach, F. Siefert, A. Knapp, and W. Reif. An Approach for Isolated Testing of Self-Organization Algorithms. In R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, editors, *Software Engineering for Self-Adaptive Systems III: Assurances*, volume 9640 of *LNCS*. Springer, 2017.

[53] B. Eberhardinger, A. Habermaier, and W. Reif. Toward Adaptive, Self-Aware Test Automation. In *Proc. of the 12th IEEE/ACM Int. Wsh. on Automation of Software Testing*, pages 34–37. IEEE Press, 2017.

[54] B. Eberhardinger, H. Seebach, D. Klumpp, and W. Reif. Test Case Selection Strategy for Self-Organization Mechanisms. In M. Winter, A. Spillner, and A. Pietschker, editors, *Test, Analyse und Verifikation von Software – gestern, heute, morgen*. dpunkt Verlag, 2017.

[55] B. Eberhardinger, H. Seebach, A. Reichstaller, A. Knapp, and W. Reif. Adaptive Tests for Adaptive Systems: The Need for New Concepts in Testing for Future Software Systems. *Softwaretechnik-Trends*, 38(1), 2017.

[56] B. Eberhardinger, H. Ponsar, D. Klumpp, and W. Reif. Measuring and Evaluating the Performance of Self-Organization Mechanisms within Collective Adaptive Systems. In *Proc. of the 8th Int. Symp. on Leveraging Applications of Formal Methods*, LNCS. Springer, 2018.

[57] B. Eberhardinger, H. Ponsar, G. Siegert, and W. Reif. Case Study: Adaptive Test Automation for Testing an Adaptive Hadoop Resource Manager. In *Proc. of the 18th IEEE Int. Conf. on Software Quality, Reliability and Security Companion*, pages 513–518, 2018.

[58] J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive Software System Monitoring for Performance Anomaly Localization. In *Proc. of the 8th ACM Int. Conf. on Autonomic Computing*, pages 197–200. ACM, 2011.

[59] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem. Runtime Verification of Component-Based Systems. In G. Barthe et al., editors, *Proc. of the 9th Int. Conf. Software Engineering and Formal Methods*, volume 7041 of *LNCS*, pages 204–220. Springer, 2011.

[60] A. Filieri, C. Ghezzi, and G. Tamburrelli. A Formal Approach to Adaptive Software: Continuous Assurance of Non-functional Requirements. *Formal Aspects of Computing*, 24 (2):163–186, 2012.

[61] A. S. Foundation. Apache Hadoop 2.9.1—Documentation. `https://hadoop.apache.org/docs/current/`, Jul 2018. Accessed: 2018-07-09.

[62] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with Model Checkers: A Survey. In *Software Testing, Verification and Reliability*, volume 19, pages 215–261. Wiley, 2009.

[63] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng. Towards Run-time Testing of Dynamic Adaptive Systems. In *Proc. of the 8th Int. Symp. Software Engineering for Adaptive and Self-Managing Systems*, pages 169–174. IEEE, 2013.

[64] E. M. Fredericks, B. DeVries, and B. H. C. Cheng. Towards Run-time Adaptation of Test Cases for Self-adaptive Systems in the Face of Uncertainty. In *Proc. of the 9th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, pages 17–26. ACM, 2014.

[65] R. S. Freedman. Testability of Software Components. *IEEE Trans. Software Engineering*, 17 (6):553–564, 1991.

[66] E. Gamma, D. Riehle, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[67] A. Goodloe and L. Pike. Monitoring Distributed Real-Time Systems: A Survey and Future Directions. Technical Report NASA/CR-2010-216724, NASA Langley Research Center, July 2010.

[68] H. Götz, M. Winter, C. Brandes, and T. Roßner. *Basiswissen Modellbasierter Test*. dpunkt.verlag, 2016.

[69] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang. An Extensible, Regular-Expression-Based Tool for Multi-Language Mutant Generation. In *Proc. of the 40th Int. Conf. on Software Engineering)*. IEEE/ACM, 2018.

[70] M. Güdemann, F. Ortmeier, and W. Reif. Formal Modeling and Verification of Systems with Self-x Properties. In *Proc. of the 3rd Int. Conf. on Autonomic and Trusted Computing*, pages 38–47. IEEE, 2006.

[71] A. Habermaier. *Design Time and Run Time Formal Safety Analysis using Executable Models*. PhD thesis, University of Augsburg, 2016.

[72] A. Habermaier, B. Eberhardinger, H. Seebach, J. Leupolz, and W. Reif. Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#. In *Proc. of the 9th IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 128–133. IEEE, 2015.

[73] A. Haddadi and K. Sundermeyer. Belief-Desire-Intention Agent Architectures. *Foundations of Distributed Artificial Intelligence*, pages 169–185, 1996.

[74] J. Hänsel, T. Vogel, and H. Giese. A Testing Scheme for Self-Adaptive Software Systems with Architectural Runtime Models. In *IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 134–139. IEEE, 2015.

[75] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-Tuning System for Big Data Analytics. In *Proc. of the 5th Int. Conf. on Innovative Data Systems Research*, volume 11, pages 261–272, 2011.

[76] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[77] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, Dec. 1990.

[78] Institute of Electrical and Electronics Engineers. *IEEE Guide to Software Requirements Specifications*. IEEE, 1984.

[79] Intel. GitHub: Intel HiBench. `https://github.com/intel-hadoop/HiBench`, Jul 2018. Accessed: 2018-07-09.

[80] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization for MapReduce Programs. *Proc. of the 37th Int. Conf. on Very Large Data Bases*, 4(6):385–396, 2011.

[81] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *Proc. of the 34th Int. Conf. on Software Engineering*, pages 1427–1430. IEEE, 2012.

[82] Y. Jin and J. Branke. Evolutionary Optimization in Uncertain Environments – A Survey. *IEEE Trans. on Evolutionary Computation*, 9(3):303–317, 2005.

[83] B. F. Jones, H. Sthamer, and D. E. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[84] E. Kaddoum, C. Raibulet, J. Georgé, G. Picard, and M. P. Gleizes. Criteria for the Evaluation of Self-* Systems. In *Proc. of the 3rd Wsh. on Software Engineering for Adaptive and Self-Managing Systems*, pages 29–38. ACM, 2010.

[85] I. Kant. *Kritik der Urteilskraft*. 2nd edition, 1793.

[86] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Proc. of the IEEE Int. Conf. on Neural Networks*, volume 4, pages 1942 –1948, 1995.

[87] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36 (1):41–50, 2003.

[88] D. Klumpp, A. Habermaier, B. Eberhardinger, and H. Seebach. Optimising Runtime Safety Analysis Efficiency for Self-Organising Systems. In *Proc. of the 1st IEEE Int. Wsh. on Foundations and Applications of Self* Systems*, pages 120–125. IEEE, 2016.

[89] C. G. Lee and S. C. Park. Survey on the Virtual Commissioning of Manufacturing Systems. *Journal on Computational Design and Engineering*, 1(3):213–222, 2014.

[90] M. Leucker and C. Schallhart. A Brief Account of Runtime Verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[91] J. Leupolz. *Probabilistic Safety Analysis of Executable Models*. PhD thesis, University of Augsburg, 2018.

[92] M. Luckey, C. Thanos, C. Gerth, and G. Engels. Multi-Staged Quality Assurance for Self-Adaptive Systems. In *Proc. of the 6th Int. Conf. Self-Adaptive and Self-Organizing Systems Wsh.*, pages 111–118, 2012.

[93] M. R. Lyu. *Handbook of Software Reliability Engineering*, volume 222. IEEE Computer Society, 1996.

[94] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: An Automated Class Mutation System: Research Articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, 2005.

[95] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1992.

[96] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[97] C. McGeoch. *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

[98] P. McMinn. Search-Based Software Testing: Past, Present and Future. In *Proc. of the 4th IEEE Int. Conf. on Software Testing, Verification and Validation Workshops*, pages 153–163, 2011.

[99] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive Testing with Asynchronous Communications and Timestamps. *Journal on Distributed Computing*, 31(5):327–342, 2018.

[100] B. Meyer. Eiffel: A Language and Environment for Software Engineering. *Journal on Systems and Software*, 8(3):199–246, 1988.

[101] J. Miller, M. Reformat, and H. Zhang. Automatic Test Data Generation Using Genetic Algorithm and Program Dependence Graphs. *Information and Software Technology*, 48(7): 586–605, 2006.

[102] W. Miller and D. L. Spooner. Automatic Generation of Floating-Point Test Data. *IEEE Trans. on Software Engineering*, SE-2(3):223–226, 1976.

[103] H. D. Mills. The Management of Software Engineering: Part I: Principles of Software Engineering. *IBM Syst. Journal*, 19(4):414–420, Dec. 1980.

[104] L. Monostori, B. C. Csáji, B. Kádár, A. Pfeiffer, E. I. Zudor, Z. Kemény, and M. Szathmári. Towards Adaptive and Digital Manufacturing. *Annual Reviews in Control*, 34(1):118–128, 2010.

[105] M. Morandini, L. Penserini, and A. Perini. Modelling Self-Adaptivity: A Goal-Oriented Approach. In *Proc. of the 2nd IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems*, pages 469–470. IEEE, 2008.

[106] L. J. Morell. A Theory of Fault-Based Testing. *IEEE Trans. on Software Engineering*, 16(8): 844–857, 1990.

[107] J. D. Musa. A Theory of Software Reliability and its Application. *IEEE Trans. on Software Engineering*, (3):312–327, 1975.

[108] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley, 3rd edition, 2011.

[109] F. Nafz. *Verhaltensgarantien in selbst-organisierenden Systemen*. PhD thesis, University of Augsburg, 2012.

[110] F. Nafz, H. Seebach, J. Steghöfer, G. Anders, and W. Reif. Constraining Self-Organisation Through Corridors of Correct Behaviour: The Restore Invariant Approach. In *Organic Computing - A Paradigm Shift for Complex Systems*, pages 79–93. 2011.

[111] K. Naik and P. Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. Wiley, 2011.

[112] J. J. Naresky. Reliability Definitions. *IEEE Trans. on Reliability*, R-19(4):198–200, 1970.

[113] J. Neyman. Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability. *Philosophical Transactions of the Royal Society London A*, 236(767):333–380, 1937.

[114] C. D. Nguyen. *Testing Techniques for Software Agents*. PhD thesis, Università di Trento, 2009.

[115] C. D. Nguyen, A. Perini, and P. Tonella. Goal-Oriented Testing for MASs. *Int. Journal of Agent-Oriented Software Engineering*, 4(1):79–109, 2009.

[116] C. D. Nguyen, A. Marchetto, and P. Tonella. Automated Oracles: An Empirical Study on Cost and Effectiveness. In B. Meyer, L. Baresi, and M. Mezini, editors, *Proc. of the 9th Joint Meet. Europ. Software Engineering Conf. and ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 136–146. ACM, 2013.

[117] Object Management Group (OMG). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, 2005. URL `http://www.omg.org/spec/QVT/1.0/PDF`. Accessed: 2018-07-09.

[118] L. Padgham, J. Thangarajah, Z. Zhang, and T. Miller. Model-based Test Oracle Generation for Automated Unit Testing of Agent Systems. *IEEE Trans. on Software Engineering*, 39(9): 1230–1244, 2013.

[119] H. V. D. Parunak and S. A. Brueckner. Software Engineering for Self-organizing Systems. In *Proc. of the 12th Int. Wsh. on Agent-Oriented Software Engineering*, pages 1–22, 2011.

[120] Personalized Medicine Coalition. *The Case for Personalized Medicine.* 4th edition, 2014.

[121] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques.* Wiley, 2008.

[122] J. Pitt, D. Busquets, and S. Macbeth. Distributive Justice for Self-organised Common-pool Resource Management. *ACM Trans. on Autonomous and Adaptive Systems*, 9(3):14, 2014.

[123] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI Reasoning Engine. In *Multi-Agent Programming: Languages, Platforms and Applications*, pages 149–174. Springer, 2005.

[124] M. Polo, P. Reales, M. Piattini, and C. Ebert. Test Automation. *IEEE Software*, 30(1):84–89, 2013.

[125] M. Popovic and J. Kovacevic. A Statistical Approach to Model-Based Robustness Testing. In *Proc. of the 14th IEEE Conf. and Wsh. on Engineering of Computer-Based Systems*, pages 485–494, 2007.

[126] A. Pretschner and J. Philipps. Methodological Issues in Model-Based Testing. In Broy et al. [21], pages 281–291.

[127] G. Püschel, S. Götz, C. Wilke, C. Piechnick, and U. Aßmann. Testing Self-Adaptive Software: Requirement Analysis and Solution Scheme. *Int. Journal on Advances in Software*, 7:88–100, 2014.

[128] T. Rahwan, S. D. Ramchurn, N. R. Jennings, and A. Giovannucci. An Anytime Algorithm for Optimal Coalition Structure Generation. *Journal of Artificial Intelligence Research*, 34: 521–567, 2009.

[129] S. D. Ramchurn, P. Vytelingum, A. Rogers, and N. R. Jennings. Putting the "Smarts" into the Smart Grid: A Grand Challenge for Artificial Intelligence. *Communications of the ACM*, 55(4):86–97, 2012.

[130] A. J. Ramirez, A. C. Jensen, B. H. C. Cheng, and D. B. Knoester. Automatically Exploring How Uncertainty Impacts Behavior of Dynamically Adaptive Systems. In P. Alexander et al., editors, *Proc. of the 26th IEEE/ACM Int. Conf. Automated Software Engineering*, pages 568–571. IEEE, 2011.

[131] R. Ramler and K. Wolfmaier. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. In *Proc. of the 1st Int. Wsh. on Automation of Software Test*, pages 85–91. ACM, 2006.

[132] A. Reichstaller and A. Knapp. Transferring Context-Dependent Test Inputs. In *IEEE Int. Conf. on Software Quality, Reliability and Security*, pages 65–72, 2017.

[133] A. Reichstaller, B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif. Applying Deep Learning For Imitating Adaptive Agent Behavior in Statistical Software Testing. *Softwaretechnik-Trends*, 38(1), 2017.

[134] A. Reichstaller, B. Eberhardinger, H. Ponsar, A. Knapp, and W. Reif. Test Suite Reduction for Self-organizing Systems: A Mutation-based Approach. In *Proc. of the 13th Int. Wsh. on Automation of Software Test*, pages 64–70. ACM, 2018.

[135] P. Reinecke, K. Wolter, and A. Van Moorsel. Evaluating the adaptivity of computing systems. *Performance Evaluation*, 67(8):676–693, 2010.

[136] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads. *Proc. of the 39th Int. Conf. on Very Large Data Bases*, 6(10):853–864, 2013.

[137] RespectIT. Objectiver Homepage, May 2018. URL `http://www.objectiver.com/index.php?id=4`. Accessed: 2018-05-09.

[138] G. Roşu. Formal Methods in System Design: A Monitor Synthesis Algorithm for Past LTL. Technical report, University of Illinois at Urbana-Champaign, 2007.

[139] G. Roşu, F. Chen, and T. Ball. Synthesizing Monitors for Safety Properties – This Time With Calls and Returns. In *Proc. of the 8th Workshop on Runtime Verification*, volume 5289 of *LNCS*, pages 51–68. Springer, 2008.

[140] L. Sabatucci, V. Seidita, and M. Cossentino. The Four Types of Self-adaptive Systems: A Meta-Model. In G. De Pietro, L. Gallo, R. J. Howlett, and L. C. Jain, editors, *Intelligent Interactive Multimedia Systems and Services*, pages 440–450. Springer, 2018.

[141] H. Samih, H. Le Guen, R. Bogusch, M. Acher, and B. Baudry. An Approach to Derive Usage Models Variants for Model-Based Testing. In M. Merayo and E. de Oca, editors, *Proc. of the 26th IFIP WG 6.1 Int. Conf. Testing Software and Systems*, volume 8763 of *LNCS*, pages 80–96. Springer, 2014.

[142] O. Sammodi, A. Metzger, X. Franch, M. Oriol, J. Marco, and K. Pohl. Usage-Based Online Testing for Proactive Adaptation of Service-Based Applications. In *Proc. of the 35th IEEE Computer Software and Applications Conf.*, pages 582–587, 2011.

[143] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter. Adaptivity and Self-organization in Organic Computing Systems. *ACM Trans. on Autonomous Adaptive Systems*, 5(3):10:1–10:32, 2010.

[144] H. Seebach. *Konstruktion selbst-organisierender Softwaresysteme.* PhD thesis, University of Augsburg, 2011.

[145] H. Seebach, F. Nafz, J. Steghöfer, and W. Reif. How to Design and Implement Self-organising Resource-Flow Systems. In *Organic Computing – A Paradigm Shift for Complex Systems*, pages 145–161. Birkhäuser, 2011.

[146] G. D. M. Serugendo, N. Foukia, S. Hassas, A. Karageorgos, S. K. Mostéfaoui, O. F. Rana, M. Ulieru, P. Valckenaers, and C. Van Aart. Self-Organisation: Paradigms and Applications. In *Proc. of the 1st Int. Wsh. on Engineering Self-Organising Applications*, pages 1–19. Springer, 2003.

[147] F. Siefert. *Selbst-organisiertes, trust-bewusstes Supply Demand Management in Smart Grids.* PhD thesis, University of Augsburg, 2017.

[148] B. R. Siqueira, F. C. Ferrari, M. A. Serikawa, R. Menotti, and V. V. de Camargo. Characterisation of Challenges for Testing of Adaptive Systems. In *Proc. of the 1st Brazilian Symp. on Systematic and Automated Software Testing*, pages 11:1–11:10. ACM, 2016.

[149] C. Smidts, C. Mutha, M. Rodríguez, and M. J. Gerber. Software Testing with an Operational Profile: OP Definition. *ACM Computing Surveys*, 46(3):39:1–39:39, 2014.

[150] Spirals-Team. GitHub: Spirals-Team Hadoop. `https://github.com/Spirals-Team/hadoop-benchmark`, Jul 2018. Accessed: 2018-07-09.

[151] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[152] J. Steghöfer, G. Anders, F. Siefert, and W. Reif. A System of Systems Approach to the Evolutionary Transformation of Power Management Systems. In *43. Jahrestagung der Gesellschaft für Informatik e.V.*, pages 1500–1515, 2013.

[153] J.-P. Steghöfer. *Large-Scale Open Self-Organising Systems: Managing Complexity with Hierarchies, Monitoring, Adaptation, and Principled Design*. PhD thesis, University of Augsburg, 2014.

[154] J.-P. Steghöfer, B. Eberhardinger, F. Nafz, and W. Reif. Synthesis of Observers for Autonomic Evolutionary Systems from Requirements Models. In *Proc. of the 13rd IFIP/IEEE Int. Symp. on Integrated Network Management*, pages 1405–1408. IEEE, 2013.

[155] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer. NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proc. of the IEEE Int. Computer Performance and Dependability Symp.*, pages 91–100. IEEE, 2000.

[156] S. Taranu and J. Tiemann. On Assessing Self-Adaptive Systems. In *Proc. of the 8th Int. Conf. Pervasive Computing and Communications Wsh.*, pages 214–219. IEEE, 2010.

[157] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[158] F. Thillen, R. Mordinyi, and S. Biffl. Isolated Testing of Software Components in Distributed Software Systems. In D. Winkler, S. Biffl, and J. Bergsmann, editors, *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering*, volume 166 of *LNBIP*, pages 170–184. Springer, 2014.

[159] S. Tomforde, J. Haehner, H. Seebach, W. Reif, B. Sick, A. Wacker, and I. Scholtes. Engineering and mastering interwoven systems. In *Proc. of the 2nd Wsh. on Architecture of Computing Systems*, pages 1–8, 2014.

[160] T. Toroi. *Testing Component-Based Systems – Towards Conformance Testing and Better Interoperability*. PhD thesis, University of Eastern Finland, 2009.

[161] M. Trapp and D. Schneider. Safety Assurance of Open Adaptive Systems – A Survey. In N. Bencomo, R. France, B. H. C. Cheng, and U. Aßmann, editors, *Models@run.time: Foundations, Applications, and Roadmaps*, pages 279–318. Springer, 2014.

[162] M. Utting and B. Legeard. *Practical Model-Based Testing – A Tools Approach*. Morgan Kaufmann, 2007.

[163] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification & Reliability*, 22(5):297–312, 2012.

[164] A. Van Lamsweerde, A. Dardenne, B. Delcourt, and F. Dubisy. The KAOS project: Knowledge Acquisition in Automated Specification of Software. In *Proc. of the AAAI Spring Symp. Series*, pages 59–62, 1991.

[165] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. A Framework for Evaluating Quality-driven Self-adaptive Software Systems. In *Proc. of the 6th Int. Symp. Software Engineering for Adaptive and Self-managing Systems*, pages 80–89. ACM, 2011.

[166] M. A. Vouk. Back-to-back Testing. *Journal on Information Software Technology*, 32(1): 34–45, 1990.

[167] W. W. Eric, G. Ruizhi, L. Yihao, R. Abreu, and F. Wotawa. A Survey on Software Fault Localization. *IEEE Trans. on Software Engineering*, 42(8):707–740, 2016.

[168] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary Test Environment for Automatic Structural Testing. *Information and Software Technology*, 43(14):841–854, 2001.

[169] E. J. Weyuker. The Evaluation of Program-based Software Test Data Adequacy Criteria. *Communications of the ACM*, 31(6):668–675, 1988.

[170] E. J. Weyuker and T. J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Trans. on Software Engineering*, (3):236–246, 1980.

[171] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J.-M. Bruel. RELAX: A Language to Address Uncertainty in Self-adaptive Systems Requirement. *Requirements Engieering*, 15(2):177–196, 2010.

[172] M. Winter. Optimale Integrationsreihenfolgen. In *Software Engineering 2013: Fachtagung des GI-Fachbereichs Softwaretechnik*, pages 259–270. 2013.

[173] F. Wotawa. Adaptive Autonomous Systems – From the System's Architecture to Testing. In R. Hähnle and et al., editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, CCIS, pages 76–90. Springer, 2012.

[174] F. Wotawa. Testing self-adaptive systems using fault injection and combinatorial testing. In *Proc. of the IEEE Int. Conf. on Software Quality, Reliability and Security Companion*, pages 305–310, 2016.

[175] J. Wu, L. Yang, and X. Luo. Jata: A Language for Distributed Component Testing. In *Proc. of the 15th Asia-Pacific Software Engineering Conf.*, pages 145–152, 2008.

[176] Y. Yao and Y. Wang. A Framework for Testing Distributed Software Components. In *Proc. of the IEEE Conf. on Electrical and Computer Engineering*, pages 1566–1569. IEEE, 2005.

[177] R. Yeh, editor. *Toward a Theory of Testing: Data Selection Criteria*, volume 2, 1977. Prentice-Hall.

[178] B. Zhang, F. Křikava, R. Rouvoy, and L. Seinturier. Self-Balancing Job Parallelism and Throughput in Hadoop. In *Proc. of the 16th IFIP WG 6.1 Int. Conf. on Distributed Applications and Interoperable Systems*, volume 9687 of *LNCS*, pages 129–143. Springer, 2016.

[179] B. Zhang, F. Křikava, R. Rouvoy, and L. Seinturier. Hadoop-benchmark: Rapid Prototyping and Evaluation of Self-adaptive Behaviors in Hadoop Clusters. In *Proc. of the 12th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems*, pages 175–181. IEEE, 2017.

[180] Z. Zhang, J. Thangarajah, and L. Padgham. Model Based Testing for Agent Systems. In Decker et al., editors, *Proc. of the 8th Int. Conf. on Autonomous Agents and Multiagent Systems*, pages 1333–1334. IFAAMAS, 2009.