André Manuel Reichstaller

Machine Learning-based Test Strategies for Self-Adaptive Systems

Dissertation zur Erlangung des Doktorgrades
an der Fakultät für Angewandte Informatik der
Universität Augsburg

Universität
Augsburg
University

2020

Erstgutachter: Prof. Dr. Alexander KNAPP
Zweitgutachter: Prof. Dr. Bernhard BAUER

Tag der mündlichen Prüfung: 31.07.2020

Abstract

With self-adaptive systems a new class of reactive software systems is recently gaining lots of attention. Able to adapt their actual approach to meet given goals at runtime based on previously gained insights such systems actually appear to be kind of artificially intelligent and able to learn. Autonomous vehicles, robots, and adaptive production plants are just a few of the instances for which practical application promises huge efficiency enhancements for industry. Apart from all the advances being made in this area there still is a blocker for practical application in critical fields: how to adequately test a system whose runtime approach is actually unknown?

As this thesis elaborates, traditional test strategies for reactive systems are not feasible anymore. Building on Harel and Pnueli's notion of a development process for reactive systems an extension for self-adaptivity as well as particular challenges and requirements for testing self-adaptive systems are derived. We will see that test strategies for self-adaptive systems should be adaptive as well. A number of experiments is reported in which Machine Learning approaches were used for solution. Considering a couple of case studies, such as a Smart Vacuum Cleaner, a Smart Energy Grid, and a Self-Organizing Production Cell, those experiments are meant to provide different aspects and possible approaches for systematically testing self-adaptive systems. Requirements and outlooks for future work are given.

Kurzfassung

Selbst-adaptive Systeme sind in der Lage ihre Strategie zur Laufzeit an gegebene Ziele und Erfahrungen anzupassen. Das lässt diese Subklasse reaktiver Systeme (künstlich) intelligent und lernbar erscheinen. Obwohl Anwendungsfälle wie autonome Fahrzeuge, Roboter, oder adaptive Fabrikanlagen den praktischen Nutzen belegen, steht der großflächige Einsatz dieser Technologie noch immer vor einer wesentlichen Hürde: wie lässt sich ein System mit unbekannter Laufzeitstrategie adäquat testen?

Vorliegende Arbeit zeigt anhand unterschiedlicher Fallstudien, dass traditionelle Teststrategien für diesen Zweck nicht ausreichen. Mithilfe Harel und Pnuelis Begriff eines Entwicklungsprozesses für reaktive Systeme und einer Erweiterung dieses Prozesses für den Fall der Selbst-Adaptivität werden Herausforderungen und Ansätze für das Testen selbst-adaptiver Systeme erarbeitet. Es zeigt sich, dass Teststrategien für selbst-adaptive Systeme selbst adaptiv sein sollten. Anhand eines zu testenden intelligenten Saugroboters, eines intelligenten Energienetzes und einer selbstorganisierenden Produktionszelle werden mögliche Ansätze unter Zuhilfenahme von Methoden des maschinellen Lernens eingeführt und evaluiert. Auf dieser Basis werden mögliche Zweige zukünftiger Arbeiten aufgezeigt.

Acknowledgements

First and foremost I am extremely grateful to my supervisor Alexander Knapp for making this work possible. Thank you for your invaluable advice, your continuous support, and your patience during my study.

I would also like to thank the team of researchers and students at the Institute of Software & Systems engineering in Augsburg for the great collaboration on the research project "Testing Self-Organizing, Adaptive Systems (TeSOS)".

I thank Lenz Belzner and Thomas Gabor from the LMU Munich for a cherished time spent together discussing and refining our research findings.

I would like to thank all the members from the former Tactico team - this student project was actually the starting point of my research in artificial intelligence.

Finally, I would like to express my gratitude to my grandparents Karin, Erich, and Marianne, my parents Katharina and Claus, my wife Nardos and my children Elijah and Noël. It would not have been possible without your tremendous understanding and encouragement.

Contents

Chapter 1

Introduction

System classification has always been of particular importance for research in software engineering. The associated hope is that system classes offer generalization in development and maintenance processes which is preventing us from continuously reinventing the wheel. For this the classification is usually based on the major challenges we face within the systems development. If a major challenge exists for the one of the systems contained in a class, at least a similar challenge should also exist for the other systems of the same class; so a solution for the first should also imply a similar solution for the others. The hereby enabled abstraction then shows itself in unified processes, software libraries or frameworks usable in practice. When considering particular challenges for particular systems, it is thus important to ask if those challenges can be generalized which might justify new system classes.

One of the most general system classifications had been formally provided by Harel and Pnueli [HP85]. Distinguishing systems which are *"relatively easy to handle from those which are more complex"* their distinction was between the so-called *transformational* and *reactive* systems. While the note that *"once the problematic part of this pair [of system classes] is satisfactorily solved, most of the others will yield less painfully too"* was certainly meant more like a big picture than a time-boxed vision we can, after decades of research, finally say that most of those problematic parts have been addressed. Model- and state-based approaches on design and engineering of requirements, object-oriented approaches on programming, and systematic, automatized methods on testing are just a few instances giving us abstraction above technical complexity in practical software engineering today. Thousands of tools and frameworks already exist.

With this success the call for digitization of nearly all business and industrial processes is getting increasingly louder, bringing along plenty of special cases and motivating the introduction of new subclasses of reactive systems. *Self-adaptive systems*, the systems I want to consider here, are one of those subclasses currently gaining particular attention. In contrast to other reactive systems, a self-adaptive systems is supposed learning and choosing the particular approach for imposed goals at runtime, letting it appear to be *artificially intelligent*. Since various challenges have been already identified and solved, the hope (and also the fear) in society grows that there will soon be a breakthrough in the practical application of such systems. And indeed, considering leading edge experiments and studies we have to notice that it has already come. Smart homes [Jen+18], autonomous vehicles [Yaq+19], and intelligent production schedulers [Was+18] are just a few of the most prominent instances.

If we consider the industrial practice, we have, however, to state that there still is a major challenge preventing this technique from being comprehensively applied: quality assurance – and especially testing of systems with the capability to take and adapt decisions at run-time. On the one hand it is pretty unclear how to show that a system the actual approach of which was not explicitly designed at design time but learned at runtime will do what we expect it to do. On the other, this remains a precondition for practical application in most industrial cases. How should we introduce self-driving cars without being sure that they will not fail and thus threaten human lives? How should we establish autonomous, self-adaptive production plants without being sure they will not fail and generate enormous costs? We must not do that.

In consequence there is a major need for systematic testing approaches for self-adaptive systems. As Kapitel 2 will show, some of the techniques addressing this challenge for reactive systems are reusable indeed. The feature of runtime adaptation, however, is demanding additional efforts (Kapitel 3). As found by several experiments and justified in [Ebe+17a] we can say that

"Adaptive systems need adaptive test strategies."

This thesis is collecting experiments I performed and published in this area. They are drawing a path through a process for systematically deriving and applying such strategies. The traditional process for testing reactive systems is extended in a way that it can cope with situations in which the expected system approach cannot be derived from design time artefacts. Instead, various machine learning (ML) techniques are used for capturing the actual system behavior from simulations of the system under test in order to derive and finally apply test policies later in reality.

1.1 Outline

The thesis is structured as follows: Kapitel 2 und 3 are setting the context and giving some formal framework. Starting with reactive systems and classical testing techniques for those in Kapitel 2, the challenges for testing self-adaptive systems are step-by-step derived from the particular properties distinguishing self-adaptive systems from other reactive systems. Kapitel 3 elaborates those properties by refining a notion of the typical development process for reactive systems in a way that it complies with common definitions for self-adaptive systems.

Kapitel 4 considers a case in which with a self-organizing production cell a distributed self-adaptive system should be tested. The previously elaborated challenges for testing self-adaptive systems are shown on this instance and a way to solve them by means of ML-based clustering techniques is proposed. For this, a novel state-based type of test models and a representativity-based coverage goal for this model are proposed.

Kapitel 5 discusses a way to introduce other prioritization techniques than pure representativity in test design. Since testing self-adaptive systems mostly involves simulations of the system under test – which is also assumed in the new testing process proposed in Kapitel 3 – it is argued that we can actually predict the effect of failures using the simulation. A mutation-based test goal is proposed that considers this anticipated effect in prioritization. Custom variants of evolutionary algorithms are used for optimizing this goal by constructing adequate test cases for the self-organizing production cell considered.

As Kapitel 6 explains, one issue with self-adaptive systems is the continuous nature of state spaces in which they are often embedded. Equivalence classes in states, as required for state-based test models such as those we considered in Kapitel 4, cannot always be formed since the system behavior is unknown. We do not know the peculiarities of the system's environment. Thus, the need for continuous types of test models is motivated. An experiment capturing adaptive agent behavior in *Artificial Neural Networks* is presented on the instance of a Smart Energy Grid; and a method for deriving test cases using the ML technique of *Autoencoding* is proposed.

Kapitel 7 considers systems still learning at test 21time. A self-adaptive *Smart Vacuum System* continuously updating its map of its environment is taken as instance. It is shown that in this case it does not suffice to capture the system behavior from a given simulation once but this needs to be continuously repeated. The test process for self-adaptive systems is modified accordingly. A technique from *Reinforcement Learning*, the so-called *Direct Future Prediction*, is shown to be particularly suitable for automatically learning and generalizing test strategies from simulation in this case.

Kapitel 8 finally discusses the issues we obtain when applying a test policy derived from simulation in reality. Since, due to features in the environments which are controllable in simulation but not in reality, it is not always possible to directly apply all the test cases found, an approach is needed for translating the test case from simulation to the fixed setting in reality. A ML-based method for transferring test inputs from an uncontrollable context to a controllable one is proposed and evaluated on the Smart Vacuum System instance.

1.2 Publication List

1. André Reichstaller, Benedikt Eberhardinger, Alexander Knapp, Wolfgang Reif, and Marcel Gehlen. *"Risk-based interoperability testing using reinforcement learning"*. In: IFIP International Conference on Testing Software and Systems. Springer. 2016, pp. 52–69.

2. André Reichstaller, Benedikt Eberhardinger, Hella Seebach, Alexander Knapp, and Wolfgang Reif. *"Applying deep learning for imitating adaptive agent behavior in statistical software testing"*. In: Softwaretechnik-Trends, 38 (1). 2017.

3. André Reichstaller, Benedikt Eberhardinger, Hella Seebach, Alexander Knapp, and Wolfgang Reif. *"Test Suite Reduction for Self-organizing Systems: A Mutation-based Approach"*. In: AST'18:IEEE/ACM 13th International Workshop on Automation of Software Test. 2018.

4. André Reichstaller, Thomas Gabor, and Alexander Knapp. *"Mutation-based test suite evolution for self-organizing systems"*. In: International Symposium on Leveraging Applications of Formal Methods. Springer. 2018, pp. 118–136.

5. André Reichstaller and Alexander Knapp. *"Hunting the Game: Towards a game of testing adaptive systems"*. In: International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE. 2016, pp. 303–304.

6. André Reichstaller and Alexander Knapp. *"Compressing Uniform Test Suites Using Variational Autoencoders"*. In: International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE. 2017, pp. 435–440.

7. André Reichstaller and Alexander Knapp. *"Transferring Context-Dependent Test Inputs"*. In: International Conference on Software Quality, Reliability and Security. IEEE. 2017, pp. 65–72.

8. André Reichstaller and Alexander Knapp. *"Risk-based Testing of Self-Adaptive Systems using Run-Time Predictions"*. In: 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO). IEEE. 2018, pp. 80–89.

9. Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, and Andreas Kyek. *"Optimization of global production scheduling with deep reinforcement learning"*. In: Procedia CIRP 72.1. 2018, pp. 1264–1269.

10. Bernd Waschneck, André Reichstaller, Lenz Belzner, Thomas Altenmüller, Thomas Bauernhansl, Alexander Knapp, and Andreas Kyek. *"Deep reinforcement learning for semiconductor production scheduling."* In: 29th Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC). IEEE. 2018.

11. Benedikt Eberhardinger, Hella Seebach, André Reichstaller, Alexander Knapp, and Wolfgang Reif. *"Adaptive tests for adaptive systems: The need for new concepts in testing for future software systems."* In: Softwaretechnik-Trends 37.3. 2017.

Chapter 2

Testing Reactive Systems

According to Harel and Pnueli *reactive systems* are systems that maintain an ongoing relationship with their environment and continuously respond to external inputs. *Transformational systems*, as the counterparts of reactive systems, "accept inputs, perform transformations on them and produce outputs" [HP85, p. 479]. Since undoubtedly each form of adaptation requires continuous interactions with the system's environment, self-adaptive systems are rather reactive than transformational. And as this is the case, we are allowed to assume that for the development of self-adaptive systems similar approaches can be used as for developing other reactive systems. This should also hold for software testing.

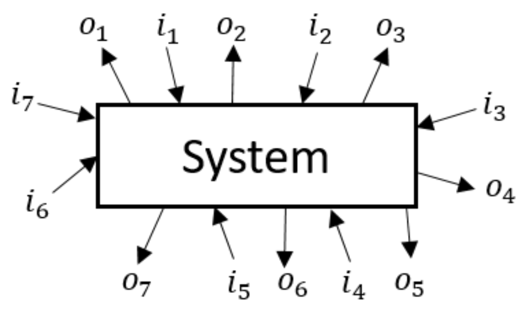
Even though we will see that there actually are additional challenges when testing self-adaptive systems requiring additional solutions, the fundamental concepts, in fact, remain unchanged. The motivation for this chapter is thus to present the basic principles of testing reactive systems in order to set the basis for elaborating the additional challenges for testing self-adaptive systems later.

The present account does not have the claim of comprehensiveness but it is meant to give the reader an impression of how reactive systems are systematically tested today.

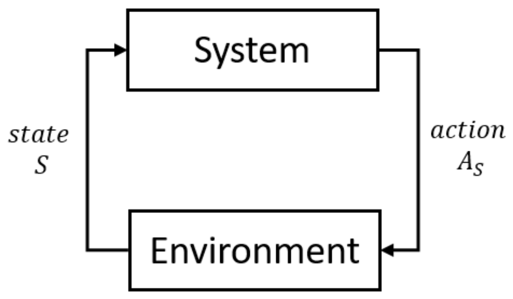
2.1 Reactive Systems

According to Harel and Pnueli's point of view the first characteristic of reactive systems is the continuous nature of interaction with an environment. This implies the importance of the dimension of time, i.e., a history of inputs and outputs, in system behavior.

Second, the interaction with an (possibly not fully predictable) environment implies a much more complex input space to be considered than a number of fixed, acceptable, inputs which are imperatively and directly transformed to outputs. There is by no means a binary borderline between complex and not-complex input spaces indeed. Further, a sequence of inputs over time might be viewed as a kind of coherent meta-input which is, of course, transformed to an output again. In fact one can say that each CPU, at the end, performs a transformation between in- and outputs, and that, in consequence, each reactive software system is also a kind of complex transformational system. But this is exactly the point: while the difference between reactive and transformational systems is not visible in an underlying computational



(A) "Black Cactus" [HP85, p. 479].



(B) Time-based generalization.

FIGURE 2.1: Interface between a reactive system and its environment.

model, and thus is not formally depictable, it is reflected in the need for translation between reactive behavioral specifications and transformational CPUs.

The following subsections consider the resulting complexity for system design and implementation in order to motivate the significant role of software testing for developing reactive systems.

2.1.1 Continuous Interaction with the Environment

Harel and Pnueli sketch the interrelation between a reactive system and its environment by what they call a "black cactus" (see Fig. 2.1a). The system processes and transforms various inputs to various outputs. Though, according to Fig. 2.1a, the major difference to transformational systems seems the sheer number of inputs and outputs, the authors also state that it is the dimension of time which makes dealing with reactive systems such complex: sensed inputs at time t might not only lead to a direct output but also influence the system behavior at $t + 1$. Technically, this dependency on time can be implemented by encapsulating historical input information as well as corresponding behavioral information in various internal system states. A reactive system is supposed to comprise multiple such internal states which it is able to switch in response to specific environmental states. The interactions between the reactive system and its environment can be thus assumed ordered in time.

To make this observation more explicit, Fig. 2.1b suggests a generalization of the cactus depiction presented: an environmental state s is assumed to comprise all the inputs sensed at logical time step t while action a comprises all the outputs in response. On this way, the system and its environment are interacting over (logical) time steps $t, t + 1, \ldots, t + n$.

A Development Process for Reactive Systems The major challenge for developing reactive systems is finding (transformational) implementations for (time-dependent) behavioral specifications which do not "(...) lend themselves naturally to description in terms of functions and transformations" [HP85, p. 479]. In contrast to transformational systems, a reactive system's implementation will not result from sequential refinements of a given behavioral specification, but it involves an additional technical process.

Harel and Pnueli conclude that "by and large, the development of a reactive system is not a one-dimensional process in which specification and design are two temporally related stages, but rather it is a two dimensional *magic square* in which they play the role of the dimensions themselves" (see Fig. 2.2) [HP85, p. 488]. They refer

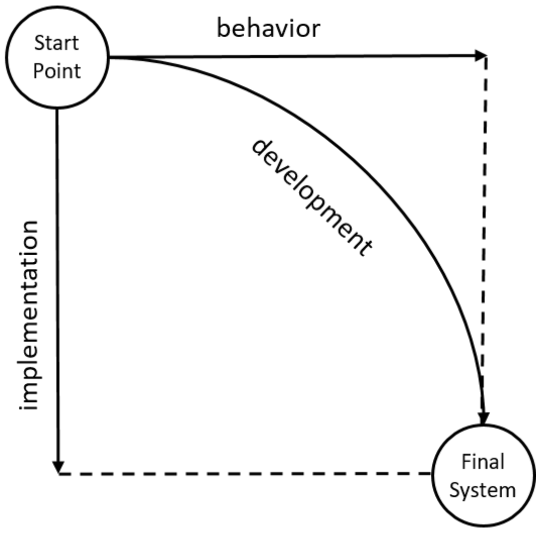


FIGURE 2.2: Harel and Pnueli's "magic square".

to a typical development process as a sequence of two-dimensional refinements:

$$(M^{(0)}, S^{(0)}) \rightsquigarrow (M^{(1)}, S^{(1)}) \rightsquigarrow \cdots \rightsquigarrow (M^{(f)}, S^{(f)}) \tag{DP}$$

Each of the pairs $(M^{(i)}, S^{(i)})$ comprise a behavioral specification $S^{(i)}$ as well as an implementational specification $M^{(i)}$ which both are getting more and more concrete with increasing i. While $(M^{(0)}, S^{(0)})$ is assumed highly underspecified, $(M^{(f)}, S^{(f)})$ describes the final system. The index i can be consequently viewed as a counter of logical time steps which is passed through the progress within the development. An implementational specification $M^{(i)}$ is further assumed to comply with an interface $E^{(i)}$ describing the inputs and outputs to be considered within the system's interaction with its environment.

2.1.2 Running Example: A Smart Vacuum System

Let us, for instance, consider developing a smart vacuum system (SVS) [FDC14; UMN97]. This little robot is supposed to drive around in order to vacuum-clean rooms of arbitrary dimensions without any human intervention. To accomplish this task, it is able to perceive its immediate surroundings by sensors, to plan routes, and to pilot its wheels in any desired direction. While its overall goal is to explore the room and to clean the dust, the SVS also has to handle dangers lurking on the way: there are obstacles located inside the room, such as furniture items or stairwells. Collisions or falls must be crucially avoided, since damage could become costly. The snippet of code depicted in Lst. 2.1 gives a first abstract view on a possible implementation.

LISTING 2.1: Abstract Simple SVS

```
public abstract class SVS {
   SensorCtr sensorCtr = new SensorCtr();
   ActuatorCtr actuatorCtr = new ActuatorCtr();

   /** This method controls the SVS behavior */
   public void drive(){
      // Get inputs for all directions
      Map<Direction,Integer> inputs = new HashMap<>();
      inputs.put(Direction.North, sensorCtr.sense(Direction.North));
      inputs.put(Direction.South, sensorCtr.sense(Direction.South));
      inputs.put(Direction.East, sensorCtr.sense(Direction.East));
      inputs.put(Direction.West, sensorCtr.sense(Direction.West));

      // Plan based on inputs
      Action action = this.plan(inputs);

      // Execute chosen action
      actuatorCtr.act(action);
   }

   /** Plan which action to execute */
   public abstract Action plan(Map<Direction,Integer> inputs);
}
```

If we merely considered the main task of cleaning dirt on its own one could say that the SVS rather is a transformational than a reactive system. This is because the SVS just has to take few sensory inputs, it has to process those inputs in a functional way, and then it has to determine the outputs. As the exemplary implementation of the SVS in Lst. 2.2 depicts, there is no dependence of the robot's behavior on a history of inputs. In this case the SVS bases the action selection on an evaluation of sensory inputs for each direction. The `Policy` is used for interpreting the calculated score. For the time being, let us assume that it deterministically takes the action with the highest score.

Let us additionally introduce some special features now. Suppose, for example that particular tasks of the SuT can be partly controlled through a mobile device, as it is in fact meanwhile usual for this kind of systems: the user is able to sketch the room the SVS will be located in within an associated mobile app, and mark positions which should be particularly taken into account for cleaning. Given such additional user inputs the SVS is supposed to immediately drive to the marked position in order to solve the task. Figure 2.3 depicts an exemplary situation.

In this case the system approach depends on the history of inputs since an output, i.e., a robot's action, at a time *t* depends on an internal state indicating whether and where a user-given mark had been set an how the room sketched by the user looks like. Let us for instance consider the implementation depicted in Lst. 2.3 making use of the Gang of Four's *state pattern* [Gam+15] for switching between the modes *Clean* and *Reaching a Mark*.

The principle idea is to avoid *switch-case* statements over an object's (in our case the SVS's) behavior, but to encapsulate its states within objects itself – and thus to conform with the *Open-Closed-Principle* described by Meyer [Mey88] [1]. A superordinate context, which in our case also acts as observer for the user's app, switches

[1]The Open-Closed-Principle states that "Modules should be both open (for extension) and closed (for modification)" [Mey88].

LISTING 2.2: Transformational Implementation

```java
public class ClassicalSVS extends SVS {
   Policy policy = new Policy();
   static final int DANGEROUS = 4;
   static final int OPT = 1;

   @Override
   public Action plan(Map<Direction,Integer> inputs) {
      // Evaluate goodness of sensor values
      Map<Direction, Integer> scores = new HashMap<>();
      for(Map.Entry<Direction, Integer> e : inputs.entrySet()){
         int input = e.getValue();
         scores.put(e.getKey(), input == DANGEROUS? Integer.MAX_VALUE:
             Math.abs(OPT - input));
      }

      // Return optimal action w.r.t. policy
      return policy.selectAction(scores);
   }
}
```

the states at runtime. In fact, the state pattern as well as the Open-Closed-Principle address in particular object-oriented system designs – and there would have been plenty of other options for implementing the system that would also conform with the specification. The chosen structure for implementation was rather motivated by technical considerations than being the result of sequential refinements of the given behavioral specification. This (technical) design dimension within the development process conforms with Harel and Pnueli's thoughts concerning the magic square (cf. Fig. 2.2).

2.1.3 A Process for Testing Reactive Systems

Besides finding an appropriate implementation, the two-dimensional development process visualized by the magic square (cf. Fig. 2.2) implies yet another major challenge to be considered: *quality assurance*. The further away the technical (implementation) is from the behavioral (specification) structure and decomposition, the harder it is assuring behavioral correctness afterwards. According to Galin [Gal04, p. 27] quality assurance includes "a systematic planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines." Let us, in the following, concentrate on assuring the functional requirements, i.e., answering the question:

Is there a match between behavioral requirements and the actual implementation?

In terms of the development process we want to make sure that each refined behavioral specification matches with its predecessor, i.e., $\forall i \in [1..f] \, . \, S^{(i-1)} \models S^{(i)}$ (*Consistency*) and that the implementational specifications match with the behavioral specifications, i.e., $\forall i \in [1..f] \, . \, M^{(i)} \models S^{(i)}$ (*Conformance*). If this is the case we also know that $M^{(f)} \models S^{(0)}$. While consistency might be supported a priori by providing behavioral specifications that are "well-structured, concise, unambiguous, readable, and easy to understand" and at the same time "solely descriptive, eliminating, or

LISTING 2.3: Reactive Implementation of SVS

```java
public class SVSContext extends SVS implements Observer{
   private SVSState svsState;
   private Point mark; private Room room;

   public SVSContext() {
      setState(new Clean());
   }

   @Override
   public Action plan(Map<Direction,Integer> inputs) {
      return svsState.plan(this,inputs);
   }

   public void setState(SVSState newState) {
      svsState = newState;
   }

   @Override
   public void update(Observable o, Object arg) {
      UserInput userInput = (UserInput)arg;
      // Store the user given inputs
      mark = userInput.getMark(); room = userInput.getRoom();
   }

   interface SVSState{
      public Action plan(SVSContext ctx, Map<Direction,Integer> inputs);
   }

   /** Implementing the SVS's clean mode **/
   class Clean implements SVSState{

      @Override
      public Action plan(SVSContext ctx, Map<Direction,Integer> inputs) {
         // Check for transition
         if(ctx.mark != null) {
            ctx.setState(new ReachMark());
            return ctx.plan(inputs);
         }
         // Plan according to classical SVS implementation
         return new ClassicalSVS().plan(inputs);
      }
   }

   /** Implementing the SVS's reach mark mode **/
   class ReachMark implements SVSState{

      @Override
      public Action plan(SVSContext ctx, Map<Direction,Integer> inputs) {
         // Check for transition
         if(position(inputs) == mark) {
            ctx.mark = null;
            ctx.setState(new Clean());
            return ctx.plan(inputs);
         }
         // calculate direction
         return path(inputs, ctx.room, ctx.mark);
      }

      /* Estimates current position based on sensory inputs*/
      private Point position(Map<Direction,Integer> sensoryInputs) {...}

      /* Calculates an action driving the SVS towards the target*/
      private Action path(Map<Direction,Integer> sensoryInputs, Room
          room, Point mark) {...}
   }
}
```

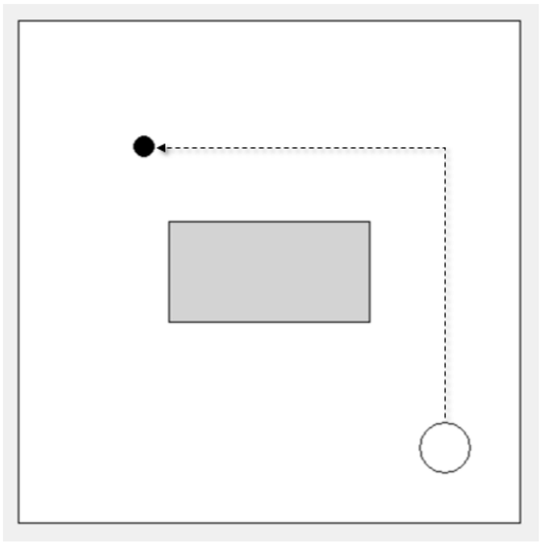


FIGURE 2.3: SVS (unfilled circle) within a room which contains an obstacle (light gray rectangle) as well as a mark (black point). The dashed arrow shows a possible path the SVS could take to the mark.

at least minimizing, dependence on any implementational issue" [HP85], assuring conformance between implementation and specification will never come around executing the final system – no matter how systematic and suitable the behavioral specification is. And exactly this is the task considered by *software testing*[2].

A Definition of Software Testing Myers et al. define software testing by "the process of executing a program with the intent of finding errors" [MSB11, p. 5]. Let me, for reasons of clarity – and without intending to dispute its general correctness, adapt that often cited definition to the more fine-grained and normed differentiation between errors, faults, and failures suggested in DIN 66271.

According to this norm, an *error* namely just denotes a human action which is leading to a fault. The *fault* itself characterizes a state of a software product which can affect the specified product function under specific conditions – and thus is a so-called *bug*. It is to note that not every fault can be revealed by executing a program under test. All we can do is to examine whether or not observed (output) values match with the given, specified ones. A deviation in those outputs would constitute a so-called *failure*.

This differentiation suggests rephrasing the definition of Myers et al. as follows:

Definition. *Testing is the process of executing a program with the intention of finding faults through revealing failures.*

As we will see, this mentioned process typically involves a whole bunch of decisions whose outcome is finally specified in so-called *test cases*: a test case specifies a particular wanted execution of a system under test (SuT) – with a set of test inputs to

[2]Note that apart from testing the functional correctness of a SuT (functional testing) as we consider it here by concentrating test effort on finding discrepancies between a SuT implementation and its behavioral specification, the literature knows lots of other testing kinds, such as stress testing [Ell86], security testing [PM04] and performance testing [WV00].

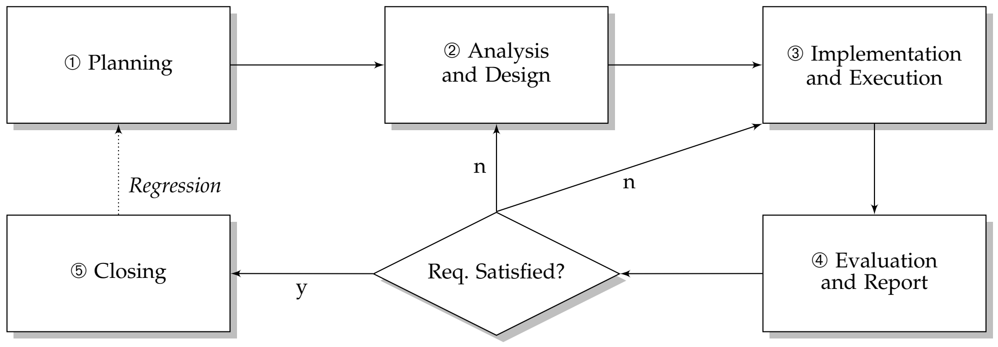


FIGURE 2.4: The traditional test process.

execute and a set of expected outputs the system is supposed to return in response to the inputs.

Due to constraints and restrictions in the SuT's execution a test case sometimes also requires some pre- and post-conditions on the environmental and/or the system state. Executing a test case then concretely means (1) to establish the preconditions, (2) to execute the SuT with the specified inputs, (3) to observe and to collect the output values, and finally (4) to study the test results comparing the output values observed with the expected ones and deciding if the system behaved correctly. A mechanism exercising (4) automatically is called a *test oracle*.

The considered Process for Testing According to Spillner et al. [SLS14] a systematical process for testing software includes, as depicted in Fig. 7.3, the following subtasks:

1. **Test Planning and Control**: First, we need to find a strategy to be pursued over the whole testing process. Such a test strategy typically involves prioritization and exit criteria for testing.

2. **Analysis and Design**: Building on this strategy and on requirement artifacts, such as behavioral specifications of the SuT, we will usually start deriving logical test cases, i.e., abstract specifications for test cases.

3. **Implementation and Execution**: Those logical test cases need to be instantiated then by concrete test cases, i.e., test cases which are executable on the SuT. Test drivers need to be established for test execution.

4. **Evaluation of Exit Criteria and Reporting**: During execution we record the time and again the exit criteria. If they are met, we will usually condense the results in kind of summary for the stakeholders.

5. **Test Closure Activities**: Finally, we evaluate the test process itself in order to make gathered experience available for future projects.

The authors mention that the subtasks may overlap and need to be adjusted to the individual needs for the particular project. As one of the main contributions of this work I will do so in Kapitel 3 for testing self-adaptive systems.

2.2 Test Strategies

Myers et al. state that the key challenge in testing is answering the following question: "what subset of all possible test cases has the highest probability of detecting the most errors?" [MSB11, p. 36]. Adapting this statement again to the DIN 66271 norm's error notion the predominant question in testing is:

What subset of all possible test cases has the highest probability of detecting the most faults by revealing failures?

In terms of the process depicted in Fig. 7.3 it is the task of test planning and test design to find an answer. Unfortunately, we usually do not know about how probable a test case (or a subset of those) reveals a failure a priori. Planning thus mostly has to start with selecting a measure which heuristically estimates the fault detection probability of arbitrary test suites. Such measures are called *adequacy criteria*. The so-called *(test) coverage* indicates the degree an adequacy criterion is met by a test suite.

Since *random testing*, i.e., testing with randomly generated test cases will usually not suffice – it was shown that the probability of revealing particular faults with random testing is microscopically small [OH96] – we usually employ more systematic methods then for finding a test suite which has a possibly high test coverage w.r.t. the chosen criterion in test design.

With the so-called *black-box* and *white-box* test strategies this section explores the common classes of approaches for both of the mentioned activities – planning as well as design. Interestingly this classification can be directly motivated from the magic square depicted in Fig. 2.2. That is, the very goal we pursue – assuring that the actual implementation $M^{(i)}$ at any iteration i matches the behavioral specification $S^{(i)}$ – can be approached starting from both of the dimensions: if we are starting from the behavior side, we examine for each unit (derived from any kind of decomposition) if it is correctly implemented. This is what black-box testing does.

Starting from the implementation side, we check for each unit (again derived from any kind of decomposition) if the behavioral specification agrees with the observed, implemented behavior. And this is exactly what white-box testing does. Both of the methods should not be seen as mutually exclusive. It is quite the opposite: the combination of both has shown to be most effective in revealing real failures [MSB11]. Still, testing actual software products we will never be able finding all of the possibly existing faults [Dij72].

2.2.1 White-Box Testing

White-box testing addresses test planning and test design from the dimension of implementation. The idea behind is constructing and assessing the test suite from an examination of the program's logic; and the goal is to cover as much of the entire logic of the program under test as it is possible. The notion of program logic to cover can thereby differ, in each case it however refers to a graph-based view on the source code of a program to test. Let me, in the following, discuss some of the most used white-box coverage criteria in the light of the code snippets introduced in Sect. 2.1.2.

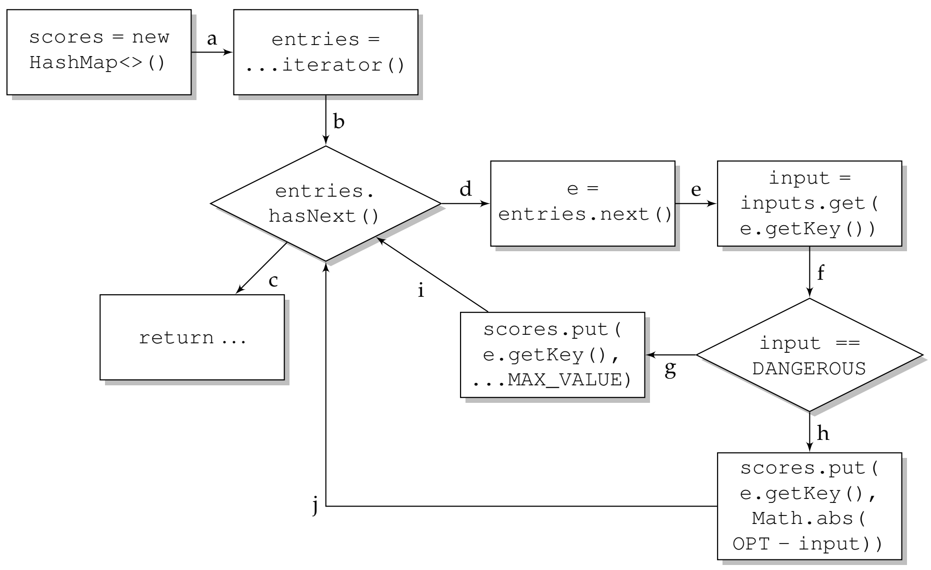


FIGURE 2.5: Control flow graph constructed from the code snipped that is depicted in Lst. 2.2. Note that some statements had been extended in order to show a more detailed view on the actual program logic.

Figure 2.5 therefor gives a graph-based representation of the transformational version of the SVS. The interested reader is referred to [KK+12] for further white-box criteria.

Statement Coverage This coverage criterion requires the test suite to cover each statement of the tested code at least once. In our example depicted in Fig. 2.5 a test case traversing the path $\langle a, b, d, e, f, g, i, d, e, f, h, j, c\rangle$ would do this job – that is e.g. the input $[\texttt{Direction.North} : \texttt{DANGEROUS}, \texttt{Direction.South} : 2]$. Even though statement coverage is generally seen as rather poor w.r.t. its fault-detection capability, it still sets the basis for the following more sophisticated coverage criteria.

Branch Coverage This coverage criterion, which is sometimes also called *decision coverage*, states that each branch direction has to be taken at least once. This means that each decision (decisions are pictured as diamonds in Fig. 2.5) will one time evaluate to *true* and one time to *false*. Apart from some rather artificial exceptions branch coverage subsumes statement coverage. In our example we actually obtain full branch coverage by the same path and test inputs as before.

Condition Coverage This coverage criterion particularly considers decisions including more than one condition. The idea is that each condition in each decision should take all possible outcomes at least once. Again there are theoretically cases in which not all the statements are covered in spite of full condition coverage. Even though in our example of Fig. 2.5 decision and condition coverage are actually the same, as each decision comprises one single condition only, in fact, condition coverage not generally subsumes decision coverage. This is usually fixed by combining the both in the so-called *decision/condition coverage* criterion that requires fulfilling both.

White-Box Test Design Given such white-box coverage criteria it is the task of test design to systematically obtain test cases that optimize those. The most simplest but, as stated before, rather ineffective answer is random testing (often also called fuzzing or monkey testing). Methods of this kind propose just randomly generated inputs for test execution [DN84; CS04; FM00]. It is easy to see that the more complex the program the more unlikely the full satisfaction of a coverage criterion through random testing. That is why several improvements evolved.

To give an instance let us consider one of the most prominent of those improvements: the so-called *concolic* testing methodology. As a portmanteau of "concrete" and "symbolic" the term concolic testing describes a test generation technique that builds on a combination of *symbolic* and *concrete execution* [SMA05]. Symbolic execution thereby means to interpret a program execution with symbolic values for inputs instead of concrete ones in normal execution. On this way, each branch within the source code can be represented by a number of constraints in terms of those symbolic values. Solving those constraints for a particular path lets us consequently find a class of concrete inputs which would trigger the system to take it.

By performing a depth-first search through a tree spanned by all the paths detected, this approach may be directly used for generating a test suite [KPV03]. In fact it is however known to be "computationally intractable" [SMA05]. This is why concolic methods proceed in a more heuristic way including elements of random search [GKS05]: first, the program to test is executed with randomly chosen concrete inputs while concurrently logging path constraints for this particular execution. The logged constraints are then used for symbolic execution trying to solve the constraints leading to a new concrete path for execution. This procedure is repeated until no additional execution path is found anymore.

Let us for instance consider testing the plan method within the non-adaptive implementation of the SVS depicted in Lst. 2.2. And let us initially guess the input $[\texttt{Direction.North} : 2]$. As a result, `plan` would execute the then-branch of the if-statement and return a selected action based on the score 1. Intertwined with the execution the predicate $\mathit{inputs}^0[\texttt{Direction.North}] \neq \texttt{DANGEROUS}$ is formed. Here, inputs^0 is the *symbolic variable* for the concrete input. By calculating a solution for $\mathit{inputs}^0[\texttt{Direction.North}] = \texttt{DANGEROUS}$ the concolic test program would then force the SuT through a different execution path, that is the else-branch of the if-statement.

2.2.2 Black-Box Testing

The black-box approach starts the test process from the dimension of behavioral specification in terms of the magic square (cf. Fig. 2.2). The idea behind is to investigate for each part of the specification whether or not it is correctly implemented. Access to source code is not assumed given. All the tester can do thus is executing the (compiled) system with selected test inputs striving to reveal some kind of failures.

Though it would be, in some cases, theoretical possible to execute the program with all possible test inputs, it is practically not. For the instance of the SVS implementation considered in Sect. 2.1.2 there would be, e.g., infinitely many different input vectors possible (as parameters are real valued).

Again, the general issue is thus selecting an appropriate subset of test cases which is supposed to maximize the probability of finding most of the faults. And again, test adequacy is formulated in terms of different kinds of criteria. This subsection summarizes the most popular ones and gives some hints on how they can be systematically achieved. The interested reader is referred to [KK+12] for further discussions about black-box testing techniques.

Equivalence Partitioning "Reasonable" testing, as Myers et al. call what is also often referred to as systematic testing, involves to minimize redundancy between the test cases to execute. The term redundancy is thereby defined on a notion of equivalence on test inputs: two test cases with equivalent test inputs are assumed to be redundant as the set of possibly revealed faults is assumed to be the same as when just executing one of the test cases.

Myers et al. conclude that equivalence classes over test inputs should be chosen in a way "(...) that you can reasonably assume (but, of course, not be absolutely sure) that a test case of a representative value of each class is equivalent to a test of any other value. That is, if one test case in an equivalence class detects an error all other test cases in the equivalence class would be expected to find the same error. Conversely, if a test case did not detect an error, we would expect that no other test case in the equivalence class would fall within another equivalence class, since equivalence classes may overlap one another" [MSB11, p. 42]. This rather theoretical object gives rise to the question:

Which classes of inputs can be viewed as equivalent?

Since for black-box testing we have no access to code, all we can do is estimate. The process of equivalence partitioning is considered thus rather heuristic and creative than deterministic. It very much depends on the tester's intention about the implemented behavior of the system under test.

Choosing the Right Inputs With the *category-partition method* Ostrand and Balcerc proposed a systematic method for deriving equivalence classes from behavioral specifications [OB88]. They suggest a systematical decomposition of the behavioral specification in order to test the resulting units then separately. The authors notice that "the process is very similar to the high-level decomposition done by software engineers", which suggests that it strives to approximate the implementation dimension in terms of the magic square – assessing if the actual implementation fits. The category-partition method builds on the following steps: First, the tester analyzes the specification and identifies functional units that can be separately tested. For each of those units the tester then identifies parameters as well as environmental conditions which are assumed to influence the function's execution behavior – those span the *categories* of inputs.

After that, the tester again divides each of the categories in partitions which are seen as kind of representative value classes triggering estimated equivalent behavior of the system under test. To give an instance let us consider the following exemplary functional unit of the SVS's specification:

"The SVS should drive through the room, cleaning the dirt while avoiding to collide with an obstacle. If the user sets a mark, the SVS is required to immediately drive to this mark."

In addition to the explicit parameters for this functional unit – the robot's and the user-given mark's position – there are quite a few supplementary relevant environmental conditions that might influence the SVS's behavior: the ground, the illumination, and the degree of dirt within the room the SVS is located in. This might result in the following partitioning of inputs, where the bold terms denote possible categories, and the not bold ones possible partitions:

- **ground**: carpet, stone, parquet.
- **illumination**: light, medium, dark.
- **dirt**: dirty, normal, clean.
- **position**: wall, central.
- **distance to obstacle**: dangerously near, not harmless, harmless.
- **mark**: opposite side of obstacle, next to the obstacle, other direction.

Given the equivalence classes, i.e., the partitions in our case, there are basically two decisions to be made: The first is choosing concrete inputs per equivalence class. Though the definition of equivalence would suggest that this choice is arbitrary, it has proven that boundary decisions, i.e., values on the border of equivalence classes, lead to a higher payoff than just taking random values inside a class [MSB11]. For instance, for partition *harmless* of category *distance to obstacle* a boundary decision would be a position right next to partition *harmless*.

The second design decision to be made by the tester is to define the combinations to take. This issue is addressed in the discipline of *combinatorial testing*. The number of all possible combinations of equivalence classes is usually far to high to create test cases for all of them. Already for our rather simple example there are $3 \times 3 \times 3 \times 2 \times 3 \times 3 = 486$ combinations.

Combinatorial Testing Besides the *all-combinations* criterion there are multiple others which have been empirically proven. Among the most often used is the t-wise coverage criterion. It requires each possible combinations of t inputs to be included in some test case of the suite [WP01]. Most popular instances include the *1-wise* (also called each-used) and *2-wise* (also called *pairwise*) [BJE94] coverage. Table 2.1 lists the combinations for pairwise coverage of the SVS specification described above. There had been also multiple extensions proposed. Cohen et al. [Coh+03] proposed the so-called *variable strength* coverage criterion which requires higher coverage among a subset of characteristics and lower coverage across the others.

TABLE 2.1: 486 variants in 9 use cases. Generated on `https://pairwise.teremokgames.com/`

ground	illumination	dirt	position	obstacle distance	mark
carpet	light	dirty	wall	dangerous	opposite side
carpet	medium	normal	central	not harmless	next to
carpet	dark	clean		harmless	other direction
stone	medium	clean	wall	not harmless	other direction
stone	dark	dirty	central	harmless	opposite side
stone	light	normal		dangerous	next to
parquet	dark	normal	wall	harmless	next to
parquet	light	clean	central	dangerous	other direction
parquet	medium	dirty		not harmless	opposite side

In our example we could for instance require that, in addition to pairwise coverage over all the categories, the test suite should include 3-wise coverage for the categories *position*, *obstacle distance*, and *mark*, since they may appear more critical w.r.t. possible failures than the others. Some further criteria base the variable strength coverage on semantic information which is added to the classes of inputs, e.g., differing between valid and erroneous values. Examples are the *t-wise valid* and the *single error* coverage criteria [GOA05]: the *t-wise valid* coverage requires every possible combination of valid values of t parameters. The *single error* coverage requires each error value of every parameter to be included in some test case in which the rest of the values are valid.

Black-Box Test Design Grindal et al. [GOA05] summarize methods for deriving test suites that are adequate w.r.t. combinatorial adequacy criteria. The summarized methods reach from fully random to completely deterministic combination strategies such as the *all-combinations* combination strategy algorithm which generates all combinations of values of the parameters in an iterative manner. In between there are methods that Grindal et al. call *non-deterministic strategies*. Those methods are combining random selections and particular heuristics or even meta-heuristics, such as genetic and ant colony algorithms. The *Automatic Efficient Test Generator* by Cohen et al. [Coh+97] implements an instance of such non-deterministic strategies:

1. Let T be the set of already selected test cases, and let C be the set of all pairs of values of any two parameters which are still uncovered by T. Collect a number of k candidate test cases by k-time iteration over the following steps:
   (a) select a parameter p and a value for p which appears in most of the uncovered pairs in C.
   (b) Randomly order the remaining parameters.
   (c) Iterate over all parameters in the order obtained in step (b) and choose a value per parameter that is included in most of the still uncovered pairs in C.

2. Insert the test case t from T_C that covers the most pairs from C in T and remove the pairs covered by t from C. If $C \neq \emptyset$ start from (1) again. Otherwise return T.

Note that the general issue of finding the test suite of minimal size still fulfilling the t-wise coverage criterion is NP-complete [WP01] – which implies that just some additional classes or parameters of inputs lead to exponentially higher costs. The algorithm sketched above reduces complexity at the expense of the potential test suite size. Cohen et al. showed that, in general, the larger the number of considered candidate test cases k per iteration, the smaller the size of the test suite [Coh+97].

2.3 Testing with Finite State Machines

As the previous section showed, the so-called white-box as well as the black-box test strategies build test design and planning on a single dimension of the magic square respectively and do not consider the other in searching for adequate tests. If we want to consider both of the dimensions within the testing process, we have to employ both of the strategies as well. Usually, we will start with black-box test design and then will subsequently augment the designed test suite with additional test cases for meeting the white-box coverage criteria as well.

In the light of the special characteristics of reactive systems (as discussed in Sect. 2.1) such an approach, however, appears rather expensive. First, the complex input space as well as the behavioral dependence of the reactive SuT on historical inputs will blow up the space of possible equivalence classes to be considered in black-box testing. Second, the structural decoupling of the SuT's implementation from its behavioral specification implies that white-box testing may not draw benefits from previously performing black-box test design. Moreover, the SuT's history dependence charges white-box test design as well as it brings along a whole bunch of additional conditions and statements in code to be covered.

To improve efficiency it has shown thus advantageous to include some technical knowledge and assumptions about the actual implementation yet within the black-box test design. One of the methods to do so is the *model-based* test design. In its classical form, model-based testing uses abstract models of the behavioral specification of the SuT, often formalized through modeling languages such as UML [OA99; Bou+07; Dai04], SysML [Hau+10; JRR15] or mathematical notations [Tre92; Tre08; LY96], for test design and execution. Several overviews over various approaches can be found in literature [DN+07; Pre05; UPL12; KL16]. Broy et al. [Bro+05] summarize methods for model-based testing of reactive systems. The presented model types there include Labeled Transition Systems (LTS) and Finite State Machines (FSM). In this section I will particularly pursue the FSM alley as it excellently highlights the nature of reactive systems – which can be seen by the fact that already Harel and Pnueli considered this formalism in [HP85].

2.3.1 Formalism

By FSM we refer to the model of a *Deterministic Mealy Machine*. This is defined by a tuple $(I, O, Q, \delta, \lambda, q_0)$ [Mea55], with

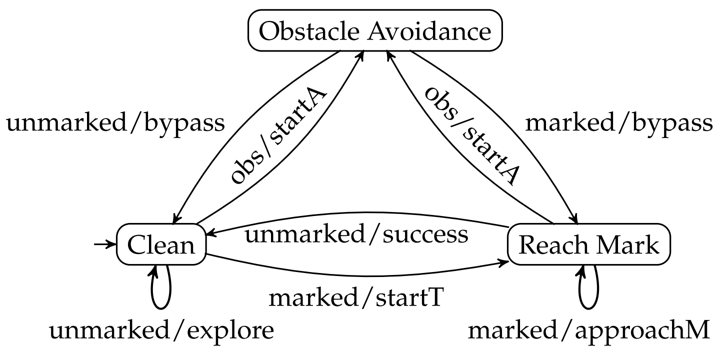


FIGURE 2.6: State diagram of an FSM describing the possible modes of the SVS.

- I a finite input alphabet;
- O a finite output alphabet;
- Q a finite set of states;
- $\delta : Q \times I \rightarrow Q$ the *transition function*;
- $\lambda : Q \times I \rightarrow O$ the *output function*;
- $q_0 \in Q$ the initial state.

Mealy machines are often depicted as state diagrams sketching transitions as arrows between states. Each arrow a connecting $s_0 \in Q$ with $s_1 \in Q$ is thereby labeled with I_a/O_a where I_a are the inputs obtained from the transition function and O_a are the outputs obtained from the output function.

Figure 2.6 depicts the state diagram of an abstract FSM for our SVS example. It models the wanted behavior of an SVS as follows: As long as no mark is set by the user and no obstacle is nearby the robot is supposed to explore its environment thereby cleaning the ground. As soon as a mark is set (input "marked"), the SVS goes in state "Reach Mark" in which it approaches the given position (output "approachM") until it is reached. Then, the robot transits again in the "Cleaning" state by signaling "success". Whenever the robot meets with an obstacle (symbolized by input "obs"), it enters the "Avoid Obstacle" state in order to "bypass" the obstacle.

As this example shows, an FSM can be seen as a formalism for specifying a system's policy denoting which action (output/state transition) the system should take in particular situations (state/input).

2.3.2 Conformance Testing

Given two FSM's M_S and M_I, conformance testing strives to examine whether M_I conforms to M_S. Faults can be detected by generating a set of input sequences from the machine M_S. The output sequences can also be obtained from M_S. This forms a test case.

The conformance property is formally defined as an equivalence (i.e., the behavior of M_I is equal to that of M_S; their initial states produce the same output for every input sequence). The input sequence for M_S that distinguishes the class of machines equivalent to M_S from all other machines is called the *checking sequence*.

In the following I will present some methods for deriving/generating such checking sequences. Those methods basically differ in the assumptions they make about M_S which leads to distinct costs (w.r.t. the test suite size and effort for test suite generation) on the one hand and a distinct fault detection capability on the other.

Assumptions First of all we have to note that for "every conformance test one can build a faulty machine that would pass such test"[Gar05]. This is because of assumptions each test has to make about the SuT. Gargantini, the author of the citation at hand, distinguishes between necessary and convenient, but not essential assumptions. The first are common for all of methods considered.

- Machine M_S is minimal, i.e., each pair of states s and t can be distinguished by an input sequence leading to different output sequences, called *separating sequence*.
- The transition and output functions are total.
- We can reach every state in M_S from every other state.
- The structure of M_I is fixed. It is able to accept all inputs in I.

There is no method which deviates from one or more of the preceding assumptions. Contrary, the following assumptions are seen as optional. Their presence can be used to classify the methods.

1. The FSMs M_S and M_I comprise the same number of states. This assumption implies that output and transfer faults can be identified through testing, but faults are not allowed to introduce new states.
2. Both FSMs accept a *reset* message in any state triggering a transition to the initial state.
3. M_I and M_S accept a *status* message in any state which leads to a response uniquely identifying the current state.
4. Both FSMs accept one *set(s)* message in the initial state for each state $s \in Q$ which lets the machines move to state s.

Methods for Test Design If all of the assumptions are given the approach of conformance testing is rather straightforward:

1. reset M to the initial state.
2. transfer M to state s through a *set(s)* message.
3. trigger transition by input message a.
4. check if received output conforms to the specified one and that the final state conforms to the specified one (by a status message).

TABLE 2.2: A possible transition tour with in- and outputs.

input	unm.	obs	unm.	marked	marked	obs	marked	unm.
output	expl.	startA	bypass	startT	appr.	startA	bypass	succ.

In most cases, however, not all of the assumptions are fulfilled. Especially the *set*-message is often not available. In this case, the literature suggests constructing a so-called *transition tour*, i.e., a sequence of inputs that visits every transition at least once [Gar05]. Table 2.2 depicts an exemplary transition tour for the instance of the SVS. If the FSM is balanced, i.e., every state has the same number of ingoing and outgoing transition, algorithms for generating an *Euler tour*, as some of them are described in [LP97], can be used for deriving the shortest transition tour possible (it visits each transition exactly once).

For unbalanced FSMs approaches based on the so-called *Chinese Postman Problem* can be applied for searching the shortest tour [EJ73]. Further approaches, such as the W [Cho78] and the Wp [Fuj+91] method, replace the *status* message with special sequences of inputs called *separating sequences* which are able to identify the source state to which they had been applied. Similar approaches additionally face the problems in which not even a *reset* message is given [Aho+91] or the implementation machine comprises a bounded number of additional states to the specification state machine [Cho78].

Chapter 3

Self-Adaptivity Changes the Game

The previous chapter considered developing – and in particular testing – the kind of systems that Harel and Pnueli referred to as being the complex ones: the so-called reactive systems. Since the time Harel and Pnueli made their observations, however, a particular sub-class of those systems evolved which is worth to be separately considered: the *self-adaptive systems*. De Lemos et al. characterize this kind of systems as "(...) systems that are able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals" [Lem+13], which implies that also this kind of systems is reactive – still there is a continuous interaction between the system and its environment.

From the functional view, developing self-adaptive systems seems actually less complex than developing other reactive systems. The more decisions the system itself takes at runtime, the less concrete decisions the developer has to take at design time concerning the wanted system behavior. However, as the saying goes: "there ain't no such thing as a free lunch" [Hei66]. The complexity reduced for functional specification is at the cost of an increased technical complexity within the implementation. Purely technical algorithms have to be included in order to enable useful, effective adaptivity. This chapter is meant to examine the implied challenges for testing self-adaptive systems.

3.1 Designing Self-Adaptive System Behavior

According to Macias et al. "[s]elf-adaptive software is capable of evaluating and changing its own behavior, whenever the evaluation shows that the software is not accomplishing what it was intended to do, or when better functionality or performance may be possible" [ME+13]. This evaluation and adaptation capability implies that self-adaptive systems actually turn the open feedback loop, in which human operators adapt a software system if necessary, to a closed feedback loop without any human intervention. Also Brun et al. find that "the feedback behavior of a self-adaptive system, which is realized with its control loops, is a crucial feature and, hence, should be elevated to a first-class entity in its modeling, design, implementation, validation, and operation" [Bru+09].

3.1.1 The MAPE-K Loop

The so-called MAPE-K loop constitutes one of the most prominent formalisms making closed feedback loops explicit. It was originally proposed by the IBM autonomic

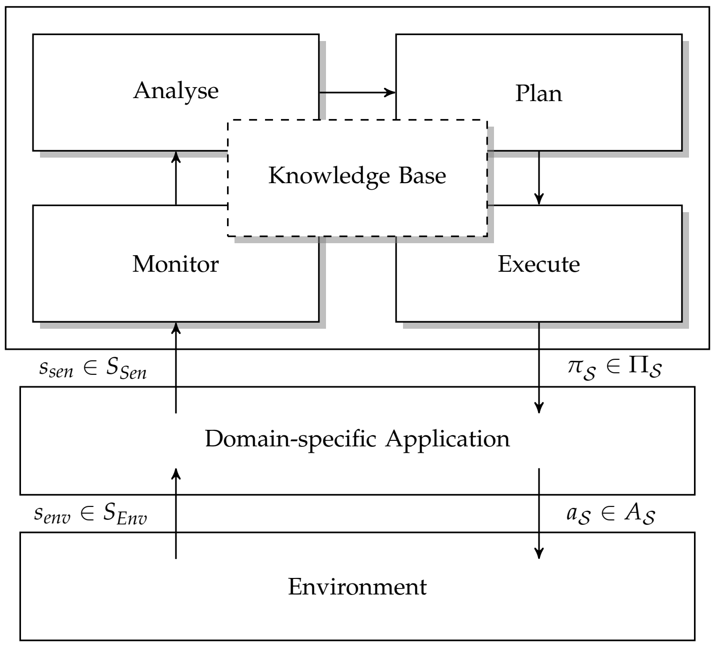

Analyse
Plan
Knowledge Base
Monitor
Execute
$s_{sen} \in S_{Sen}$
$\pi_{\mathcal{S}} \in \Pi_{\mathcal{S}}$
Domain-specific Application
$s_{env} \in S_{Env}$
$a_{\mathcal{S}} \in A_{\mathcal{S}}$
Environment

FIGURE 3.1: The MAPE-K feedback loop for self-adaptive systems. A managing system (top) is able to adapt the policy of a domain-specific managed system (middle) that continuously interacts with its environment.

computing initiative for designing so-called *self-managing* systems, i.e., systems that "(...) accomplish their functions by taking an appropriate action based on one or more situations that they sense in the environment" [Com+06]. According to the authors such self-managing systems should comprise a *managed* as well as a *managing* system component (cf. Fig. 3.1). While the managed component is usually seen as kind of resource, the managing system component is meant to replace human operators for managing it.

For self-adaptive systems, whose self-managing capabilities let them change their behavior at runtime, the managed system component can be viewed as a classical reactive system: The system continuously interacts with its environment by sensing states $s_{Env} \in S_{Env}$ and executing actions $a_{\mathcal{S}} \in A_{\mathcal{S}}$. From the FSM-based perspective each $a_{\mathcal{S}} \in A_{\mathcal{S}}$ results in an output and in a possible internal state transition. Let us assume an internal state space S_{Sys} and that the managed system in fact pursues a policy $\pi_{\mathcal{S}} : S_{Env} \times S_{Sys} \rightarrow A_{\mathcal{S}}$ that decides about how to respond to particular inputs.

In contrast to other reactive systems, however, the managed system shall be able to change its behavior at runtime. Suppose thus that actually a whole set of policies Π_{Sys} is implemented and that the single policy $\pi_{\mathcal{S}} \in \Pi_{\mathcal{S}}$ to be pursued at a given time can be switched. One could say that the implemented policies are spanning the *capabilities* of the overall self-adaptive system.

The managing system or, as it is also sometimes called, the *controller* is meant to reason about and to perform the actual policy switches at runtime. To do so it is assumed to have access to sensory data S_{Sen} comprising the perceived parts of S_{Env} as well as of S_{Sys}. More precisely, the sensed data is accessed and filtered by a *monitor* process. A so-called *analyzer* then compares this filtered data against the knowledge base and system requirements in order to diagnose possible needs for further action. If action is required, a *planner* devises a plan for adapting the managed system's policy in a profitable way. The actual policy switch is performed by effectors that are embedded within an *executor* component.

3.1.2 Implications for the Development Process

De Lemos et al. note that the traditional software development process does not meet the requirements of self-adaptive software [DL+13] anymore. They argue that "[a] self-adaptive software system operating in a highly dynamic world must adjust its behavior automatically in response to changing environments or requirements while shifting the human role from operational to strategic. Humans define adaptation goals and new application or domain requirements, and the system performs all necessary adaptations autonomously at runtime" [DL+13].

To introduce the mentioned shift of the human role as well as the system's autonomy at runtime in terms of a system design suggested by MAPE-K, the classical development process introduced in (DP) needs to be refined as follows:

$$(M^{(0)}, S^{(0)}) \rightsquigarrow \cdots \rightsquigarrow (M^{(f-2)}, S^{(f-2)}) \rightsquigarrow \left| \begin{array}{l} \overbrace{(M_{Ctr}^{(f-1)}, S_{Ctr}^{(f-1)})}^{\text{Controller}} \\ \hdashline (M_{Cap}^{(f-1)}, S_{Cap}^{(f-1)}) \rightsquigarrow \left| \begin{array}{c} \\ \underbrace{\begin{array}{c} (M_{Cap}^{(f)_1}, S_{Cap}^{(f)_1}) \\ \hdashline \vdots \\ \hdashline (M_{Cap}^{(f)_n}, S_{Cap}^{(f)_n}) \end{array}}_{\text{Capabilities}} \end{array} \right. \end{array} \right. \tag{DP*}$$

Instead of the sequential refinement of the behavioral as well as the implementational specifications until a final step f, the process is eventually split up into two different parts which may be addressed in parallel: developing the *controller* and developing the *managed system*.

While the controller is usually not subject to domain-specific specifications but is rather generic, the managed system, on the other hand, holds the domain-specific implementation by a set of concrete policies spanning the system's *capabilities*. Which of the implemented policies will be actually taken at some point in runtime is unknown at this stage. This lack of predictability of the runtime system approach at design time leads to the impression that the system is acting autonomous or proactive.

Note that runtime exchange of policies requires all policies $\{M_{Cap}^{(f)_i} \mid i \in [1..n]\}$ to implement the same interface $E^{(f)}$.

3.1.3 Capabilities

Let us, as a continuation of the running example from Sect. 2.1.2, consider a partial specification for an adaptive robot vacuum cleaner, as it is promoted at `https://www.philips.com.sg/c-p/FC8820_01`. The capabilities of this robot for cleaning a room are described by the following policies:

"4 cleaning modes to adapt to different areas in your home. The z-type mode drives the robot in a parallel zig-zag pattern, when it discovers a relatively larger space. The bounce mode drives the robot in a straight line, when it bumps into an object the robot will choose another random direction. With the spiral mode the robot moves in a spiral motion with an increasing radius. The wall-following mode drives the robot to clean while staying close and parallel to the wall."

In terms of the development process depicted in (DP*) this snippet of behavioral specification constitutes $S_{Cap}^{(f-1)}$ as it is spanning the space for possible implementations M_{Cap}^{f-1} and specifications $\{S_{Cap}^{(f)_i} \mid i \in [1..n]\}$.

Implementation Listing 3.1 shows a possible implementation of policies $\{M_{Cap}^{(f)_i} \mid i \in [1..n]\}$for this instance making use of the Gang of Four's *Strategy Design Pattern* [Gam+15]. Similar to the state pattern used for implementing the different modi of the reactive SVS in Lst. 2.3 the strategy pattern suggests to implement each policy by a separate class. In order to make sure that all policy implementations indeed follow the same interface of inputs and outputs as required before, all of the strategy classes implement the interface `CleaningStrategy`. Method `setStrategy(...)` lets us switch the policy from outside the class.

Constructing an FSM Knowing about the specification as well as about the structure of implementation, we are able to describe the managed system's behavior by means of a *finite state machine* (FSM, cf. Sect. 2.3).

In fact we are able to employ a very modular two-step modeling approach for doing so: First, we model the behavior for each possible policy separately. Each of the policy FSMs should use equal input and output functions while transitions and states may change. Second, we formalize the adaptation mechanism connecting the policy FSMs by means of an FSM as well. The overall result is a hierarchical state machine (cf. [Yan00]) with policies as nodes and transitions in between.

Figure 3.2 depicts an exemplary transition of such an hierarchical FSM in terms of the considered SVS instance. While both modelled policies comprise equal input ($I = \{wallN, wallE, wallS, wallW\}$) and output alphabets ($O = \{driveN, driveE, driveS, driveW, switchE, switchS, switchW, switchN\}$), as the figure shows, the transition functions differ.

3.1.4 Managing System

Up to now we have ignored the managed system's control: the implementation (cf. Lst. 3.1) as well as the FSM (cf. Fig. 3.2) assumed control from outside. While one

LISTING 3.1: Adaptive Implementation of SVS

```java
public class AdaptiveSVS extends SVS {
   private CleaningStrategy strategy;

   public AdaptiveSVS(CleaningStrategy strategy) {
      this.setStrategy(strategy);
   }

   public void setStrategy(CleaningStrategy strategy) {
      this.strategy = strategy;
   }

   @Override
   public Action plan(Map<Direction,Integer> inputs) {
      return strategy.plan(inputs);
   }

   interface CleaningStrategy{
      Action plan(Map<Direction,Integer> inputs);
   }

   class BounceMode implements CleaningStrategy{
      Direction currentDirection;
      Random rand = new Random();
      static final int WALL = 3;

      public BounceMode(){
         this.currentDirection = Direction.values()[rand.nextInt(4)];
      }

      public Action plan(Map<Direction,Integer> inputs) {
         if(inputs.get(currentDirection) != WALL)
            return new Action(currentDirection);
         else{
            List<Direction> others = new ArrayList<>();
            for(Direction dir : Direction.values()) {
               if(!dir.equals(currentDirection))
                  others.add(dir);
            }
            return new Action(others.get(rand.nextInt(3)));
         }
       }
    }

   class ZTypeMode implements CleaningStrategy{
      ...
      public Action plan(Map<Direction,Integer> inputs) { ... }
   }
   ...
}
```

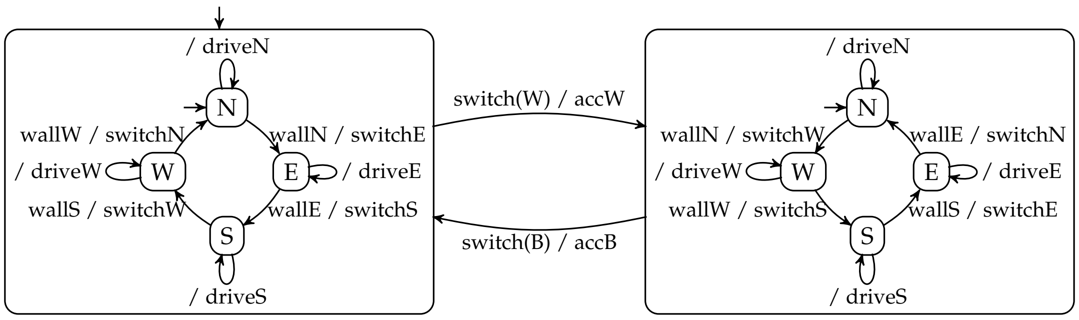


FIGURE 3.2: Exemplary transition between the *Bounce* (left) and *Wall-following* mode (right) of the SVS within a hierarchical state machine. Signals `switch(W)` and `switch(B)` for mode switches are assumed sent from outside, e.g., from the user. The outputs `accB` and `accW` are meant to signal the mode transitions.

could certainly imagine to transfer control to a user, just as an open-looped classical reactive system would, within a self-adaptive system this loop is closed by a software component: the managing system. Its predominating task is adapting the managed system at runtime. Let us in the following consider the development of such a controller. Within the process depicted in (DP*) this is represented by $(M_{Ctr}^{(f-1)}, S_{Ctr}^{(f-1)})$.

Implementation A snippet of specification representing $S_{Ctr}^{(f-1)}$ for the SVS that is promoted at `https://www.philips.com.sg/c-p/FC8820_01` reflects the need for a kind of automated feedback loop in the following way:

"The new robot vacuum cleaner is equipped with the Smart Detection System, a combination of smart chips, up to 25 sensors, gyroscope and accelerometer, that makes the robot efficient to clean on its own. ***The robot understands the environment and chooses an optimal cleaning strategy to clean your home as quickly as possible****. The robot does not get jammed and returns to its docking station when necessary."*

While it indeed requires the robot to choose the optimal cleaning policy for given environments it does not give a mapping between room types and optimal policies. The actual choice is left to the robot "itself", which in principle addresses a technical algorithm, at runtime. The exemplary implementation depicted in Lst. 3.2 delegates this choice to an abstract controller module `Reasoner`.

It sequentially updates a knowledge base with current impressions delivered by the sensors (method `updateKnowledgeBase()`) and chooses the policy to take based on this knowledge base (method `chooseStrategy()`). This might, for instance, hold a map of the room the robot is located in, together with past routes as well as estimated obstacle and dirt positions, in this way enabling a better and more efficient planning of future actions to take.

The system approach as a Markov Decision Process (MDP) One can think of diverse possible implementations for method `chooseStrategy()` within the abstract controller. A possibly simple rule-based planner might be merely equipped

LISTING 3.2: Self-adaptive Implementation of SVS

```
public class SelfAdaptiveSVS extends SVS {
   private Reasoner reasoner;
   private CleaningStrategy strategy;

   public SelfAdaptiveSVS(Reasoner reasoner) {
      this.reasoner = reasoner;
   }

   @Override
   public Action plan(Map<Direction,Integer> inputs) {
      reasoner.updateKnowledgeBase(inputs);
      strategy = reasoner.chooseStrategy();
      return strategy.plan(inputs);
   }

   interface Reasoner{
      public void updateKnowledgeBase(Map<Direction,Integer> inputs);
      public CleaningStrategy chooseStrategy();
   }

   interface CleaningStrategy{
      Action plan(Map<Direction,Integer> inputs);
   }

   class BounceMode implements CleaningStrategy{
      ...
      public Action plan(Map<Direction,Integer> inputs) { ... }
   }

   class ZTypeMode implements CleaningStrategy{
      ...
      public Action plan(Map<Direction,Integer> inputs) { ... }
   }
   ...
}
```

with static conditions (within the knowledge base) switching over environmental peculiarities extracted from the sensed states $s_{Sen} \in S_{Sen}$.

In case of the considered SVS some of those are actually mentioned in the above cited specification, e.g., *large space* $\rightarrow$ *ZigZag Pattern*. In case of such an implementation, we can obviously treat the adaptive system just like any reactive system further on – each policy can be just considered as a mode or state just like implemented in Lst. 2.3 and designed in Fig. 2.6.

Defining those static conditions, however, requires the developer to have anticipated all the possible environments with all their peculiarities the SVS could be located in at design time. If this classification turns out to be not appropriate, the system approach at runtime might fail w.r.t. the system goals. Instead of those fixed rules, it is thus reasonable to equip the SVS with the capability to classify its currently perceived environment at runtime *itself* based on previously gained experience. As this kind of implementation emphasizes the system's *self-** property against the traditional reactive nature, I refer to this as a purely *self-adaptive* implementation.

In fact, the controller's process of analyzing the inputs and planning possible policy switches for the managed system poses a *sequential decision process*, or more precisely, a *Markov Decision Process (MDP)* [FS12].

In its classical form, an MDP is represented by a tuple (S, A, T, R), where

- S is a set of states;
- A is is a set of actions;
- $T : S \times A \rightarrow PD(S)$ is a transition function which defines the effect of actions on the state of the environment with $PD(S)$ denoting the set of discrete probability distributions over S;
- $R : S \times A \times S \rightarrow \mathbb{R}$ is a reward function assigning rewards to each state transition.

Since the controller has to decide which implemented policy to choose according to the currently perceived state $s_{Sen} \in S_{Sen}$, it might be seen as instantiating the classical MDP by $(S \leftarrow S_{Sen}, A \leftarrow \Pi_{\mathcal{S}}, T \leftarrow T_K, R \leftarrow R_K)$, where

- S_{Sen} is the finite set of states that might be perceived;
- $\Pi_{\mathcal{S}}$ is the finite set of implemented policies of the managed system that can be chosen in principal;
- $T_K : S_{Sen} \times \Pi_{\mathcal{S}} \rightarrow PD(S_{Sen})$ with $T_K(s_{Sen}, \pi_{\mathcal{S}})$ indicating a probability distribution over states S_{Sen} that might result from chosing $\pi_{\mathcal{S}}$ in sensed state s_{Sen}. Note that the particular transition function is associated to the state $k \in K$ of the knowledge base as it has to be either guided by a user-given model or by self-made experience in the past;
- $R_K : S_{Sen} \times \Pi_{\mathcal{S}} \times S_{Sen} \rightarrow \mathbb{R}$ denotes rewards for taking particular transitions. Since transitions depend on the knowledge base state $k \in K$ also the reward function does.

Given one T_K and R_K the associated task is to find a (meta-)policy $\hat{\pi} : S_{Sen} \rightarrow \Pi_{\mathcal{S}}$, i.e., a rule for selecting an implemented policy in any given state that maximizes the expected return (in terms of collected reward).

Reinforcement Learning In [SB98], Sutton and Barto summarize the class of so-called *reinforcement learning* methods which are designed for solving MDPs. Algorithm 1 shows a general interface for such methods in pseudocode.

The idea behind is learning optimal policies mapping environmental states S to actions A by exploration. The exploration procedure is guided by a function `choose` that involves the current environmental state as well as a training model V in action selection. Such a training model often constitutes a value function that is mapping states and actions to expected returns according to future reward payouts. There might be however other implementations of range $\mathcal{V}$ of training models, too.

After each step of exploration the training model is updated with respect to the gained insights, that are the reached state s' as well as the obtained reward r. The learning component is usually not required to have full knowledge about the environmental dynamics i.e, the transition function T of the underlying MDP. Instead, it is merely assumed having access to an executable environment $E_T : S \times A \rightarrow S$ with $E_T(s, a)$ returning a target state s' in response to executing a in s. The selection of s' is assumed to comply with the probability distribution $T(s, a)$.

Algorithm 1 Abstract Interface of Reinforcement Learning Algorithms

```
Require: $(S, A, T, R) \equiv$ underlying MDP
         $\mathcal{V} \equiv$ range of training models storing insights gained through training
         choose $\equiv S \times \mathcal{V} \rightarrow A$ guiding exploration
         update $\equiv S \times A \times \mathbb{R} \times S \times \mathcal{V} \rightarrow \mathcal{V}$ updating the training model
         $E_T \equiv S \times A \rightarrow S$ with $E_T(s, a)$ drawing an $s'$ according to $T(s, a)$
         $nb\_steps \equiv$ number of training steps to be performed
1  $s \leftarrow$ any source state from $S$
2  $V \leftarrow$ any training model from $\mathcal{V}$
3  $step \leftarrow 0$
4  while $step < nb\_steps$ do
5      $a \leftarrow$ choose$(s, V)$
6      $s' \leftarrow E_T(s, a)$
7      $r \leftarrow R(s, a, s')$
8      $V \leftarrow$ update$(s', a, r, s, V)$
9      $s' \leftarrow s$
10     $step \leftarrow step + 1$
11 return $V$
```

As soon as a predefined number of training steps is reached, the training procedure is finished. A function like `chooseStrategy` from Lst. 3.2 might then implement a (meta-)policy by just selecting the best-rated action per state with respect to V.

Recall that the controller's MDP instantiates T and R by T_K and R_K. Thus, as the knowledge base's state might change over time by executions of method `updateKnowledgeBase` also T_K and R_K change. Since this invalidates V as well the planning procedure will have to be started again and again after each knowledge base change for calculating suitable policies.

Diverse reinforcement learning implementations are accessible with open sources and can be easily plugged in for given domain-specific problems (consider for instance *keras-rl* for deep reinforcement learning [Pla16]). The fact that using such implementations lets us simply plug-in generic pieces of software for solving domain-specific problems at runtime thereby emphasizes the benefits of self-adaptive systems again: they are saving cost and time for designing domain-specific solutions.

Implications for the FSM If we could assume given a specified controller's meta-policy $\hat{\pi} : S_{Sen} \rightarrow \Pi_{\mathcal{S}}$ we would be able to state the wanted, overall system behavior in response to each input $s_{Env} \in S_{Env}$ at design time. We just had to insert s_{Env} into the chosen policy $\pi_{\mathcal{S}}$ to obtain the system action $a_{\mathcal{S}} \in A_{\mathcal{S}}$. In particular, we could in this case specify the actual conditions leading to policy switches within the hierarchical state machine depicted in Fig. 3.2.

As discussed before the meta-policy of a purely self-adaptive system is, however, computed at runtime and its computation considers current states of the knowledge base. Further, the system is assumed to learn over time, i.e., the knowledge base's content is not assumed to be static and known a priori. If we want to predict the specified overall system behavior for a particular input at a particular time we thus not only have to reverse engineer the runtime planning procedure but also the historical

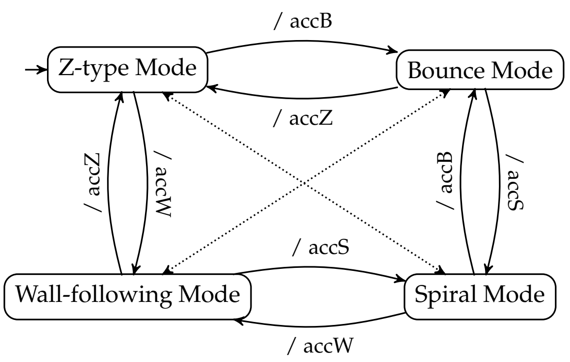


FIGURE 3.3: Non-deterministic finite state machine (NFSM) modeling the SVS's behavior. The NFSM still is hierarchical organized: each mode is assumed to include an FSM itself that indicates the SVS behavior under the respective policy.

updates of the knowledge base. Doing so at design time would require predicting all the possible environmental peculiarities the system might be confronted with at runtime. This is not possible still. In consequence, we have to assume that neither any developer nor any tester will be able to characterize the overall system behavior at design time deterministically, e.g., by indicating conditions for policy switches in the FSM.

At least, we are able to characterize a kind of superset of possible system approaches by explicitly introducing non-determinism within the behavioral models. *Non-deterministic Finite State Machines (NFSMs)* present one of the possible formalisms to do so. By replacing the transition and output functions δ and λ of a deterministic FSM described by $(I, O, Q, \delta, \lambda, q_0)$ with a so-called *transition relation* $\tau : Q \times I \rightarrow 2^{Q \times O}$, an NFSM that is described by (I, O, Q, τ, q_0) is able to model multiple transitions with diverse outputs per source state and input. Which of the possible transitions is actually taken given a particular input and state is supposed unknown a priori. Figure 3.3 depicts such an NFSM for our SVS running example in which the wanted $\hat{\pi}$ is supposed to be unknown.

3.2 Testing Self-adaptive Systems

Having discussed the particular characteristics of self-adaptive systems, let us now consider testing those systems. Recall that self-adaptive systems are reactive as well. Assuring their functional quality should consequently involve both: examining conformance between the implementation and specification as well as consistency between the respective refinements.

More specifically, we usually strive to show that $\forall i \in [1..n] \,.\, M_{Cap}^{(f)_i} \models S_{Cap}^{(f)_i}$ (conformance of the capabilities), that $M_{Ctr}^{(f-1)} \models S_{Ctr}^{(f-1)}$ (conformance of the controller), and that $\sum_{i \in [1..n]} S_{Cap}^{(f)_i} + S_{Ctr}^{(f-1)} \models S^{(f-2)}$ (consistency), i.e., that the intertwining between the managed and the managing system's specification matches with the latest common behavioral specification.

Consistency is traditionally not shown by testing but rather informally assured since pure specifications are not executable. However, the split-into-two development process for self-adaptive systems changes this situation. Informally examining consistency would namely involve to understand the interplay of specifications $\sum_{i\in[1..n]} S_{Cap}^{(f)_i} + S_{Ctr}^{(f-1)}$ at design time. If we would, one of the main purposes for using self-adaptive systems I discussed in Sect. 3.1.2 – "shifting the human role from operational to strategic" [DL+13] – was questionable. If we however do not, we need to shift the consistency check to runtime.

This means that showing consistency turns out to be a testing task also. After conformance testing we additionally strive to show that $\sum_{i\in[1..n]} M_{Cap}^{(f)_i} + M_{Ctr}^{(f-1)} \models S^{(f-2)}$, i.e., that the integrated implementation of the managed and the managing system conforms with the overall behavioral specification, by testing.

3.2.1 Conformance Testing

Testing conformance for self-adaptive systems in general has to involve both: we have to test whether the managed system implements the capabilities correctly, i.e., $\forall i \in [1..n]\,.M_{Cap}^{(f)_i} \models S_{Cap}^{(f)_i}$; and we have to test whether the managing system correctly implements the feedback loop, i.e., $M_{Ctr}^{(f-1)} \models S_{Ctr}^{(f-1)}$. As we noticed before that usually huge parts of the managing systems in fact are standardized and can be included through code bibliographies, let us assume in the following that its conformance is already tested (see [Ebe+17b] as an example where a newly developed controller is tested separately). We can thus concentrate on testing $\forall i \in [1..n]\,.M_{Cap}^{(f)_i} \models S_{Cap}^{(f)_i}$.

Testing with NFSMs Test design, i.e., selecting and prioritizing logical test cases, for conformance testing traditionally involves interpretations about the SuT's intended behavior. Deterministic finite state machines (FSM) are one way to do so (cf. Sect. 2.3). More precisely, the FSM-based conformance testing approach assumes that both the specification as well as the implementation can be represented by FSMs M_I and M_S respectively. The task is showing equivalence between the two.

We are, however, not able to state deterministic FSMs for specifications of integrated self-adaptive systems. Also the actual implementation may be of non-deterministic nature due to its dependence on possibly changing states of the knowledge base. Same inputs might lead to different outputs over time. Either way the classical FSM-based approach on conformance test design considered in Sect. 2.3 can be not directly applied. Still, the literature knows several solutions including the following instances:

With the so-called *generalized Wp method* Luo et al. adapt a traditional FSM-based testing approach to testing *equivalence* between two NFSMs [LBP94]. Both, the specification as well as the implementation are supposed to be non-deterministic. They are considered equivalent if all possible input sequences result in equal sets of possible output sequences. To obtain the sets of possible outputs from the implementation the authors suggest to execute the same input sequences a "sufficient" number of times. The assumption that all possible outputs to an input will be eventually observed is called the *complete testing assumption*.

In terms of self-adaptive systems that assumption would mean that the SuT will eventually have tried out each implemented strategy if we just wait long enough, which in the context of strategy switches appears very time-consuming.

Shabaldina et al. propose to test conformance of two NFSMs with respect to a so-called *separability relation* [SEFY07]. As this approach is getting along without the complete testing assumption it seems more suitable for testing self-adaptive systems than the one before. The idea behind is trying to show that there actually is an input sequence for which the sets of output responses of the NFSMs to the sequence do not intersect. In case such a sequence is found, there is no equivalence relation between the two NFSMs. An algorithm is given that systematically searches for such sequences.

Though the authors mention that each derived test case needs to be applied only once in order to detect every separable implementation they also note that the number of test cases derived is associated with the number of non-deterministic transitions in the specification and the implementation to be separated. A reported experiment in which an NFSM comprising 5 states, 3 inputs, and 3 outputs resulted in 15407 test cases to execute indicates high costs for using this method for testing self-adaptive systems.

In sum we have to state that though both considered methods can cope with non-deterministic specifications and implementations, they appear unfeasible in practice due to the high associated costs.

Instrumenting the Knowledge Base for Test Input Generation A starting point for limiting those high costs with respect to the number of test cases to be applied is *making* the implementation deterministic if it is not already. Recall that the non-determinism in self-adaptive system behavior comes from the managing system component. Since our primary aim here is testing the managed component however, we are allowed to just exclude the possible cause for non-determinism – that is the knowledge base.

Moreover, the knowledge base itself can serve as a kind of test input generator. Hänsel et al. argue in this context that the state of the knowledge base (which they call architectural runtime model) usually mirrors the relevant parts of sensed environmental and internal states and thus can serve as test input [HVG15]. Planning is thereby decoupled from sensors. Controlling the knowledge base we can thus on the one hand suppress further learning and, on the other hand, directly generate test inputs. Since the meta-policy $\hat{\pi}$ in this case remains fixed over time the observed system behavior becomes deterministic.

Understanding the relation of knowledge base states and $\hat{\pi}$ we can actually make use of the traditional FSM-based conformance testing methods again: we just augment the NFSM depicted in Fig. 3.3 with conditions associated to the knowledge base to obtain a deterministic FSM. Consider Fig. 3.4 for an instance.

But even if we do not understand the relation between knowledge base states and $\hat{\pi}$, complexity decreases by a deterministic implementation. Using the *generalized Wp method* by Luo et al. derived test cases for deterministic implementations need to be applied only once – the *complete testing assumption* is obviously fulfilled.

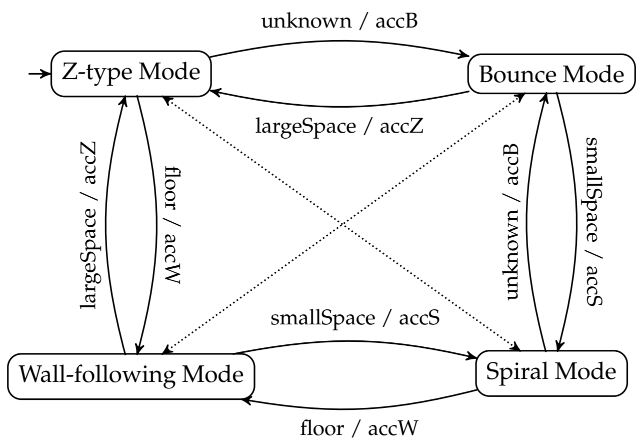


FIGURE 3.4: FSM indicating the SVS's behavior with respect to a particular knowledge base state for which the tester knows which kind of room would be classified as including, e.g., a *large space*.

As also Petrenko et al. [PYB96] suggest: the previously considered *equivalence relation* between two NFSMs can be relaxed to a *reduction relation* that requires that the set of output sequences returned by the implementation FSM can be also produced by the specification NFSM in response to every input sequence. In terms of testing self-adaptive systems such a reduction relation means that the policies the SuT follows are actually specified.

3.2.2 Testing Consistency

Having assured conformance and therefore that the capabilities themselves, i.e., the possible runtime policies the system might follow, are implemented correctly, we still do not know whether the policy selection at runtime is actually conforming with the overall system goals usually formalized in S^{f-2} [1]. The system is meant to fulfill those imposed goals at runtime by the interplay of implementations $\sum_{i\in[1..n]} M_{Cap}^{(f)_i} + M_{Ctr}^{(f-1)}$.

For testing consistency we hence cannot avoid integrating the managing and the managed systems into a single whole. Further, testing self-adaptation against environmental dynamics certainly requires to introduce dynamics in test inputs also. We need to consider sequences of environmental states $\{S_{Env}^{i}\}_{i\in\mathbb{N}}$ over time steps i as inputs further on. The issue is to make sure that for each of those possible input sequences the system behavior comprising all the steps from updating the knowledge base to responding with actual outputs conforms with the imposed goals.

[1]Consider the so-called *Corridor of Correct Behavior (CCB)* as instance for such goal formalization. This will be introduced later in the context of *self-organizing systems*, that is a subclass of self-adaptive systems.

Traditional Test Strategies The fact that we have rather to consider sequences of test inputs instead of single ones suggests that testing self-adaptive systems in fact is more complex than testing other reactive systems[2].

In terms of *concolic testing* (cf. Sect. 2.2.1) this can be seen at the complexity of the path constraints to be solved which increases with the number of inputs to be considered. Also the absolute number of paths to be tested increases due to integration of the system components. In total we will observe longer test times for achieving lower test coverage than before.

In terms of combinatorial black-box testing (cf. Sect. 2.2.2) the inclusion of time steps, i.e., the i in the sequence, in test inputs blows up the space of possible equivalence classes to be considered. Expanding, for instance, the categories and partitions chosen in Sect. 2.2.2 for testing the reactive SVS for just one additional time step we obtain $486 \times 486 = 236\,196$ combinations to be considered. The more inputs to combine, the higher, however, the amount of time needed for algorithms used to combine them – and the longer the time needed for test execution afterwards.

Applying prioritization, such as suggested by the *variable strength* coverage criterion (cf. Sect. 2.2.2), over time steps i lacks of knowledge about the intended system approach at runtime: If we do not know which are the principal characteristics that should guide the system's behavior, we are also not able to prioritize them manually. This can be also seen at FSM-based testing. Since we do not know about the intended system behavior at runtime we can also not state concrete input sequences that trigger particular paths through the FSM at design time. Eventually reaching some highly prioritizes transitions hence requires to try out all the possibilities.

Testing in Simulation In spite of the huge effort needed for deriving and executing test cases, imposed constraints in time and cost still limit the possible invest. As we saw in the previous sections the possibilities for reducing the number of test cases are limited by non-determinism. Thus, one usually tries to reduce the overall testing costs by cheapening the test execution itself. This is shifted from slow and hard-to-control reality to fast and easy-to-control simulation. If we consider, for instance, the SVS described above, it is certainly cheaper to confront it with diverse room settings in simulation than actually placing the robot again and again by hand in reality.

Various approaches have been proposed for testing self-adaptive systems in simulation. Let us in the following consider some instances: Nguyen et al. use a given environmental simulation to evolve increasingly demanding environmental setups a SuT could be confronted with by using genetic algorithms [Ngu+09]. Habermaier et al. propose to use executable environmental models for simulating test inputs at testing of self-organizing systems [Hab+15]. A model checker systematically explores the system behavior in response to each modeled environmental scenario.

A similar simulation-based approach is used for testing a self-organizing production cell in [Ebe+16]. Eberhardinger et al. consider a method for testing reconfiguration mechanisms, which may be viewed as particular instances of feedback

[2]As the previous chapter showed also not self-adaptive reactive systems require sequences of inputs to be tested. Since self-adaptive systems are assumed able to perform more than one policy, where the behavior under each of those policies forms a reactive system itself, we need to construct sequences of sequences of test inputs in this case for exhaustive testing.

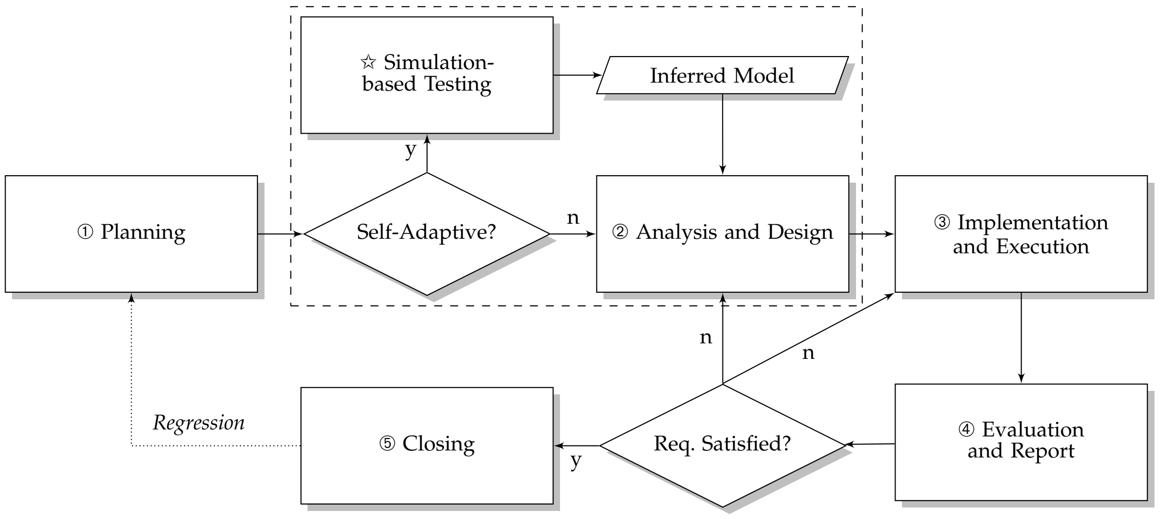


FIGURE 3.5: The traditional test process extended with Simulation-based Testing.

loops, in isolation [Ebe+17b]. The environment is thereby simulated by so-called *environmental profiles* that are supposed to be manually modeled by means of Markov Chains. The environment is thereby simulated by so-called *environmental profiles* that are supposed to be manually modeled by means of *Markov Chains*.

Fredericks et al. test the implementation of an autonomous smart vacuum system within a simulated environment using an approach for runtime test suite adaptation they call *Veritas* [FDC14]. This approach implements the idea to base testing on a feedback loop similar to MAPE-K – letting also the test component monitor, analyze and plan the following test execution [FRC13].

Hänsel et al. propose to provide a simulation for what they call *in-the-loop* testing of self-adaptive systems [HVG15]. This simulation is assumed to implement transitions between sensed states and can be thus used for generating meaningful input sequences.

3.3 A Modified Test Process

Hänsel at al. mention that the simulation itself might diverge from the actual environmental conditions and that consequently the SuT should be tested also in real, not-simulated environments [HVG15]. This statement is further motivated by literature about testing embedded software systems, where the testing process traditionally comprises three stages: first, the behavioral specification in terms of executable models is tested in simulation; second, the real software is tested in simulation; and third, the real software is tested within a real environment [BN03].

3.3.1 The Need for Reduction

The more realistic the test environment the more expensive, however, test execution. While simulations are mostly easy to access and to automatize, reality is often not. In consequence we cannot execute all test cases from simulation in reality, too.

Let us consider an instance: In [Ebe+16] Eberhardinger et al. reported about executing 131 000 test cases for testing a particular setting of a self-organizing production cell in simulation. In their setup executing all those test cases within a central simulation took 570 minutes. Since self-organizing systems are distributed, it is reasonable to assume that also their interoperability will have to be tested within a later stage. This will have to involve a distributed test environment rather than a central simulation. If we assume that setting up this environment before test execution just needs 10 seconds per test case, executing all the test cases then takes about 21 833 minutes which is about 15 days.

We obviously need to harmonize the number of test cases to execute with imposed constraints in time and cost. Though, as seen before, underspecification in the SuT's runtime behavior sets limits to the traditional approaches for doing so. Simply put, as long as we do not know about the concrete *intended* system behavior we are not able to form and to prioritize equivalence classes over possible test cases. On the other hand, the employed simulation itself lets us observe the *actual* runtime behavior of the SuT– at least for fixed knowledge base states.

Each executed test case will result in an explicit system response that might instantiate the previously non-deterministic model at a particular part since it is giving us a hint about the actual system behavior which was unknown at design time. If this is the case, the sum of all system responses obtained through executed test cases in simulation could instantiate whole parts of the previously non-deterministic behavioral model by deterministic transitions. At least for those instantiated parts we are able to apply traditional techniques for systematic test design afterwards.

3.3.2 Testing Self-adaptive Systems as a Game

For testing consistency of self-adaptive systems both appears reasonable thus: first, a simulation-based testing stage lets us cheaply explore the actual behavior of the SuT. For this, existing methods including those described in the previous subsection might be used. Second, the insights gained from simulation can be used for more targeted testing afterwards. To do so we need to capture the observed SuT behavior in any kind of abstract model. Figure 3.5 visualizes the implications for the traditional test process.

Testing in simulation can be thereby viewed as a *stochastic game* between the tester and the SuT [RK16]. While the tester is trying to reveal as many and as serious deviations from the system goals as possible, the SuT seeks to prevent exactly this through policy switches.

An instance of this *Game of Testing (GoT)* (cf. Fig. 3.6) is described by the tuple $(S_{Env}, A_{\mathcal{T}}, A_{\mathcal{S}}, T, R_{\mathcal{T}}, R_{\mathcal{S}})$, where $A_{\mathcal{T}}$ is the set of actions the tester is able to take, transition function $T : S_{Env} \times A_{\mathcal{T}} \times A_{\mathcal{S}} \rightarrow PD(S_{Env})$ defines the effect of actions on the environmental state with $PD(S_{Env})$ denoting the set of discrete probability distributions over S_{Env}, and reward functions $R_* : S_{Env} \times A_{\mathcal{T}} \times A_{\mathcal{S}} \times S_{Env} \rightarrow \mathbb{R}$ are quantifying the goals of the tester ($* = \mathcal{T}$) and the SuT ($* = \mathcal{S}$). While the SuT's goal is specified in $S^{(f-2)}$, the tester's is usually the inverse, since the ultimate goal in testing is still to reveal failures (cf. Sect. 2.1.3).

Assuming discrete time steps $t = 1, \dots, N$, the SuT and the tester are sequentially selecting actions according to their respective (meta-)policy $\pi^t_* : S_{Env} \rightarrow A_*$, which returns the action $a_* \in A_*$ to execute in a state $s^t \in S_{Env}$. At the end of each time step the environmental state s^{t+1} is updated by a state drawn from $T(s^t, a_{\mathcal{T}}, a_{\mathcal{S}})$.

The tester as well as the SuT are rewarded by $r_*^{t+1} = R_*(s^t, a_{\mathcal{T}}, a_{\mathcal{S}}, s^{t+1})$. Both are allowed to update their policy based on this obtained reward then. For the SuT this is the task of the managing system component (cf. Sect. 3.1.4).

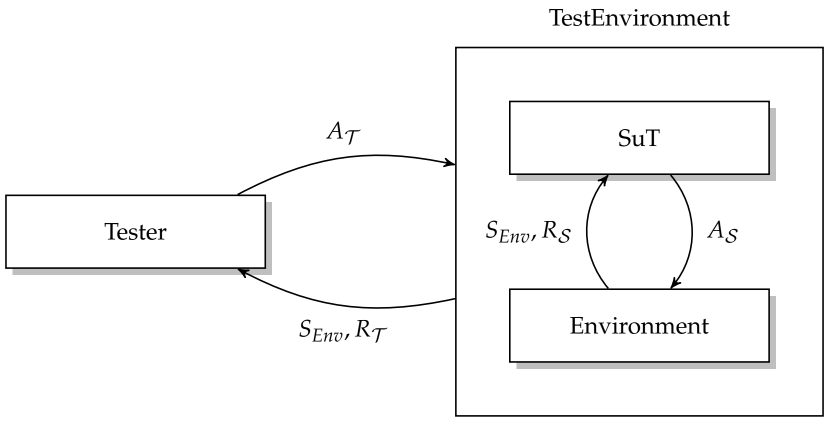


FIGURE 3.6: General setting of the Game of Testing combining two decision processes, one for the SuT and one for the tester.

But also the tester's perspective on the GoT formally cuts out a *Markov Decision Process* (MDP) for a given SuT policy $\pi_{\mathcal{S}}^t$. This MDP is of the form $(S_{Env}, A_{\mathcal{T}}, T_{\mathcal{T}}, R)$, where S_{Env} and $A_{\mathcal{T}}$ can be directly taken from the GoT and the transition function $T_{\mathcal{T}} : S_{Env} \times A_{\mathcal{T}} \rightarrow PD(S_{Env})$ as well as the reward function $R : S_{Env} \times A_{\mathcal{T}} \times S_{Env} \rightarrow \mathbb{R}$ are formed by embedding $\pi_{\mathcal{S}}^t$ within the test environment: $T_{\mathcal{T}}(s, a_{\mathcal{T}}) = T(s, a_{\mathcal{T}}, \pi_{\mathcal{S}}^t(s))$ and $R(s, a_{\mathcal{T}}, s') = R_{\mathcal{T}}(s, a_{\mathcal{T}}, \pi_{\mathcal{S}}^t(s), s')$ for $s, s' \in S_{Env}$ and $a \in A_{\mathcal{T}}$. Note that both, the transition as well as the reward function can be extended to the SuT's meta-policy $\hat{\pi}$ as this defines the policy switches at runtime.

3.3.3 Learning and Adapting Test Policies at Runtime

In the light of the GoT simulation-based test design can be viewed as trying to derive an optimal policy $\pi_{\mathcal{T}}$ w.r.t. imposed test goals from simulation that is then used for testing in later stages of the test process. When examining solutions for deriving such policies we have to distinguish whether the self-adaptive SuT itself is still learning at runtime or not[3]. In terms of the SuT's MDP the meta-policy $\hat{\pi}$ of a still learning SuT will namely change over time while this of a not-learning SuT remains fixed.

If it is fixed, an optimal policy w.r.t. the tester's MDP cannot become suboptimal again. We are able to derive an integrated model from simulation once in order to base policy selection for further testing on this. I call this activity *test policy learning*. Still, it can be advantageous to let the test policy evolve over various iterations of the

[3]Note that also the behavior of a not learning self-adaptive system may be non-deterministic. Consider for example systems with implicit feedback loops such as self-organizing systems involving emergent behavior: in order to understand their behavior at design time one has to understand the interplay of components at runtime.

test process in order to include the insights of previous test results in further testing. I will reference this approach as *test evolution* hereinafter.

The meta-policy of a still learning SuT changes over time, i.e., there is a natural number n for each time step t for which $\hat{\pi}^t \neq \hat{\pi}^{t+n}$. Since the system's policy is part of the tester's environment, an optimal test policy w.r.t. particular goals might thus become suboptimal later on. Searching for one particular policy able to perform well over time would consequently require to consider all possibly chosen meta-policies of the SuT. In other words, we would have to consider game theory: we try to learn a strategy against an opponent which is able to switch its strategy in response to the own strategy as well.

Chapter 4

Test Policy Learning

Present chapter reports about a case study of simulation-based test design for a self-adaptive system that we published in [Rei+18]. We considered the task of devising a policy for testing a *self-organizing* system (see Sect. 4.1) which serves with a particular interesting characteristics: while the system is not learning at runtime – its knowledge base and thus its meta-strategy is fixed – its intended, overall meta-policy is, still, supposed unknown at design time. To put it in the words of conformance testing we dealt with testing a deterministic implementation against a non-deterministic behavioral specification.

Indeed we rather considered testing consistency than conformance striving to show that the integrated system components are meeting the overall system goals. Therefore we assumed a simulation of the system's environment given. As the SuT's meta-policy does not change over time we performed what I previously called test policy learning (cf. Sect. 3.3.3): we derived an explicit behavioral model from simulation once in order to base test policy selection for further testing on this.

A certain type of behavioral model allowed us to integrate model-based with particular white-box considerations[1] (see Sect. 4.2 and Sect. 4.3). Given this model we regarded test policy learning as a reduction task (see Sect. 4.4): we strove for finding a subset of all the test case sequences we applied in simulation such that the chosen sequences were still representative for the rest. In terms of a test policy as it is seen in Sect. 3.3.3 we thus neglected the order in execution of the chosen test case sequences. For performing the representative subset selection two clustering techniques from machine learning literature, the *Affinity Propagation* and the *Dissimilarity-based Sparse Subset Selection*, proved particularly suitable.

The reader should note that some parts of this chapter are direct citations from paper [Rei+18] which are set in context of present work.

4.1 Testing Self-Organizing Systems

Self-organizing (SO) systems are distributed systems that are able to adapt their internal structure at runtime to changing environmental conditions without human supervision; see Fig. 4.1 for the fundamental setup. For the purpose of our case study we considered a self-organizing production cell under test. This has been

[1]Recall that, in general, traditional white-box approaches on test design are considered unfeasible for testing consistency for self-adaptive systems (cf. Sect. 3.2.2)

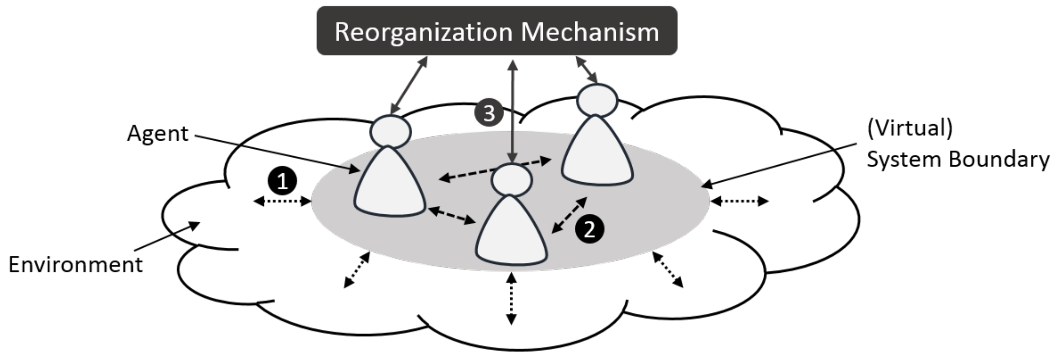


FIGURE 4.1: Fundamental setup of self-organizing systems: Distributed software agents (often embedded in physical machines) continuously interact with their environment (denoted by arrow type ❶). The joint system behavior is strongly influenced by inter-agent communication (denoted by arrow type ❷) through message passing. As soon as an agent detects an environmental fault that hinders the current system approach it triggers a reorganization mechanism (denoted by arrow type ❸) which computes and distributes a valid configuration again.

originally introduced by Güdemann et al. [Güd+08] as an example of using self-organization for mastering future production scenarios. The motivation behind was that the authors expected that production systems will have to support decentralized decision making, the optimization of their system structure, and autonomous reaction to component failures at runtime increasing the system's robustness – and the concept of self-organization promised solutions.

4.1.1 Self-Organizing Systems

As specific instances of self-adaptive systems the distributed nature of SO systems is reflected not only in the managed system – this comprises a number of distributed components or agents – but also in the controller. At least some parts of the implemented feedback loop (cf. Sect. 3.1.1) are decentralized:

The distributed agents are *monitoring* and *analyzing* their environment. If disturbances are detected, the agents report their insights to a so-called *reorganization mechanism* that can be either central or distributed again. This mechanism is then *planning* possible adaptations of the system structure which are broadcasted in the *execution* stage among the agents. Since each of the agents is assumed to perceive its environment and to act in accordance to an internal state at runtime also the overall system's *knowledge base* is distributed. In particular, each agent $ag \in Ag$ follows a local policy π_{ag} at each time that maps environmental states S_{Env} to local outputs A_{ag}. An overall system policy $\pi_{\mathcal{S}} \in \Pi_{\mathcal{S}}$ is therefore the combination of local policies of all comprised agents Ag within the system. The action space $A_{\mathcal{S}}$ is formed by the cross-product of all local action spaces.

Even though SO systems traditionally do not modify their knowledge base at runtime their explicit intended behavior is still not assumed predictable at design time. This is mainly because of two reasons: First, explicitly devising an overall system policy $\pi_{\mathcal{S}}$ by combining specifications of all local policies is usually very

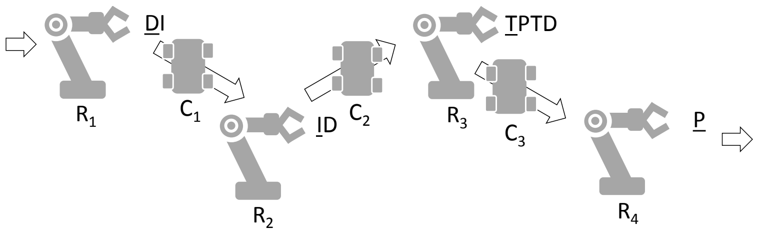


FIGURE 4.2: A schematic overview of the self-organizing production cell case study with four robots as well as three carts establishing the resource flow between them. The task is to apply the drill, insert, tighten, and polish capabilities to all incoming workpieces. Each robot's available tools are shown to its right, with D, I, T, and P; the currently allocated ones are underlined.

complex since message passing and timing need to be considered. Second, the local policies of agents can be exchanged by reorganization at runtime. Predicting the single agents' local policies at one time requires, thus, having understood the system's meta-policy $\hat{\pi}_S$ that is mapping sensed states to new *configurations* comprising a local policy per agent.

As the reorganization mechanism's specification is rather of technical than of domain-specific nature – which is in line with the assumptions concerning the managing system of a self-adaptive system (cf. Sect. 3.1.4) – constructing such a domain-specific mapping is most times practically unfeasible.

Just think about having to manually infer the equivalence classes established by, say, an ant-colony algorithm over domain-specific states. Just as for self-adaptive systems usual (cf. Sect. 3.2.2), assuring the consistency of SO systems is most times reduced thus to a testing task.

4.1.2 The Self-Organizing Production Cell

With a self-organizing production cell we considered in our case study a particular instance of an SO system under test. The production cell comprises robots (R$_1$, R$_2$, R$_3$, R$_4$) and carts (C$_1$, C$_2$, C$_3$) which are meant to process and to transport ingoing workpieces in accordance to predefined tasks. Each of those entities is equipped with a software agent controlling its capabilities through roles that implement the local policies. The robots' capabilities (Drill, Insert, Tighten, Polish) correspond with the kind of tools they are equipped and the mobile carts are able to carry the workpieces in between.

Though the production cell can in principle fulfill any task that corresponds to capabilities available, particular environmental faults such as broken tools or obstructed routes might make particular tasks unfeasible at runtime. In case of such an environmental fault the reconfiguration mechanism is meant to modify the overall system configuration in a way that present tasks can still – or again – be accomplished.

Nafz suggests to implement the single agent behavior within the self-organizing production cell with respect to the hierarchical state machine depicted in Fig. 4.3 [Naf12]. Starting in state idle the agent can be triggered by a message produce to choose and apply a role (such as controlling a driller) on a given resource res. If the

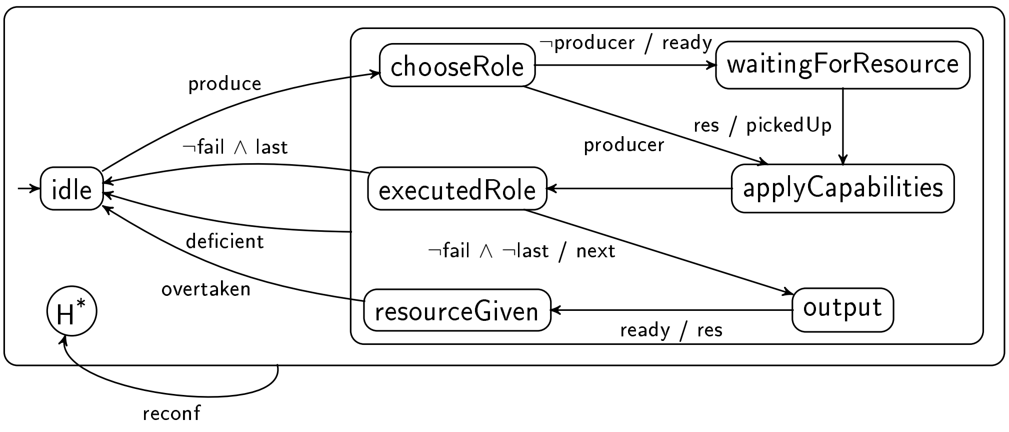


FIGURE 4.3: Single agent behavior in the adaptive production cell

agent depends on another and thus is not a so-called producer, it waits until res is available and can be pickedUp. The capabilities of the controlled robot are getting applied then. If no failure was monitored and if there is no more other robot in the resource flow, the agent immediately changes to state idle again. If there is another robot, the agent outputs a message to trigger the next. As soon as the next robot is ready, the resource res is passed to wait for the final acknowledgment message overtaken.

On top of this rather straight forward resource flow implementation, each agent is able to react on reconfiguration signals, that is, reconf and deficient, at any time. The first is broadcast by the reconfiguration mechanism if an environmental fault led to a failure and a new configuration has to be computed. As long as this message is sent, the agent stays in its current state (as indicated by the deep history pseudo-state). As soon as the deficient signal is broadcasted, however, the agent immediately changes then to state idle for restart.

The overall behavior of the self-organizing production cell results thus, on the one hand, from the distributed, parallel execution of the agents with their message exchanges (this forms a policy $\pi_S \in \Pi_S$) and the reconfiguration mechanism using reconf and deficient on the other (this forms the meta-policy $\hat{\pi}_S$).

4.1.3 The Corridor of Correct Behavior

For goal specification of the SuT we considered a formulation by Güdemann et al. that is based on the so-called *Restore Invariant Approach* (RIA) [Güd+08]. This suggests to describe the system goals, i.e., $S^{(f-2)}$ in the development process (cf. (DP*)), by means of a so-called *Corridor of Correct Behavior* (CCB). The idea behind is stating the wanted properties of internal and external states with logical predicates – the conjunction of these predicates forms an overall invariant for the system's specified run-time behavior. As soon as this invariant is broken, i.e., one of the predicates evaluates to false, the reorganization mechanism is assumed to restore the invariant through reorganization again.

In case of the self-organizing production cell scenario, Güdemann et al. suggested the following invariant: each robot needs to be equipped with all the capabilities to perform the tasks given by the allocated roles. If needed tools are determined broken, reorganization searches for a valid role allocation again.

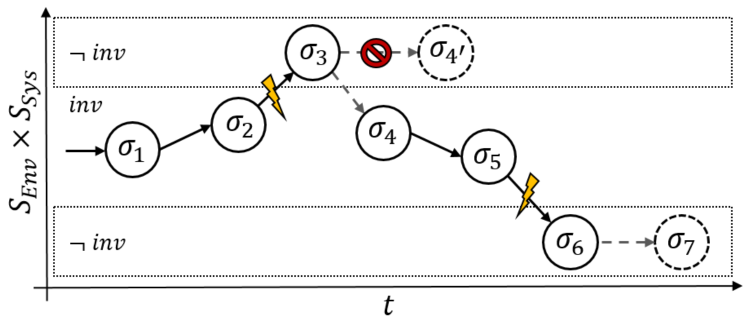


FIGURE 4.4: Corridor of correct behavior

To obtain a feeling about how the CCB at runtime should be interpreted, let us consider an informal instance depicted in Fig. 4.4. We see that the overall system state changes at runtime due to environmental dynamics. At some time the invariant is getting broken – say because some unforeseen disturbance occurred. Now, the system is meant to trigger reorganization in order to establish a configuration that, in combination with the environmental state, is located inside the corridor again.

If no such configuration is eventually established even though it would be generally possible, a failure within the integrated implementations occurred. On the other hand, the CCB does not specify which particular configuration within the corridor should be chosen – the concrete intended system behavior is at design time non-deterministic.

As discussed in Sect. 3.2.2 this non-determinism in intended behavior, however, limits test design: while white-box techniques, such as symbolic execution and concolic testing, suffer from the fact that whole sequences of environmental states need to be considered as inputs for testing the SuT against the CCB (which can be seen at the corridor depicted in Fig. 4.4), model-based testing techniques for non-deterministic behavioral models considered in Sect. 3.2.1 do not allow for prioritization of particular execution paths. Also pure black-box testing in terms of combinatorial testing suffers from both, the complex input space as well as insufficient possibilities for prioritization.

4.1.4 Simulation-based Testing using S#

In order to harmonize the needed test effort with imposed constraints in time and cost, testing an SO system's consistency is typically shifted to fast and easy-to-control simulation (c.f. Sect. 3.2.2).

Habermaier et al. propose the S# framework for this, which was originally intended for the safety analysis of component-oriented systems [Hab+15; HLR15]. With its domain-specific language which was designed on top of C#, this framework allows for modeling executable environments in which the executables controlling the agents and reorganizations can be embedded. So-called *environmental faults* can be defined which can be set on and off while simulation.

Eberhardinger et al. build on that and describe techniques for test case generation using a built-in model checker [Ebe+16; Ebe+17c]. All of those techniques basically rely on systematically activating and deactivating environmental faults in a way that possibly many reorganizations are triggered by possibly few executions, and can thus be seen as kind of combinatorial testing.

Klumpp et al. further propose some heuristic optimizations limiting the number of test cases to execute [Klu+16]. Still, the reduced number of test cases depends on the size and complexity of the environmental models defined and would be typically far too high for testing in more realistic, distributed environments.

In regards to our case study we modeled a setting of the self-organizing production cell that is described by Fig. 4.2 in S#. We defined 55 different kinds of environmental faults, such as a broken driller for specific robots. Further, we embedded three different SO mechanism implementations within the models: a handcrafted central mechanism, a mechanism based on constraint solving, and a decentralized mechanism. The combinatorial, heuristic testing approach by Klumpp et al. resulted in 7524 test cases to be executed in simulation.

4.2 The Model to Learn

As Sect. 3.3 argued before a system's consistency should be not exclusively tested within a simulation. There are usually types of faults which can only be detected in more realistic environments. For testing the agents' communication in the self-organizing production cell, for instance, we will never get around testing the SO system within a distributed environment. This real environment, however, is much harder and costlier to control than an S# simulation.

It seems thus gainful to investigate methods complementing combinatorial black-box testing with model-based or code-based test design techniques subsequent to test execution in simulation. It obviously depends on the use case which size is considered reasonable and a customizable prioritization technique hence would be preferable.

Within our case study we found that the simulation runs can be interpreted to build a finite state machine (FSM) representing the behavior of the SuT forming a, though rather partial, model for model-based testing (Sect. 4.2.1). Further, we proposed a code-based extension of this model that lets us practically include also white-box considerations in test design afterwards.

4.2.1 The Considered Model

Recall that the behavior of a self-adaptive system can in principle be modeled by means of a hierarchical FSM (cf. Sect. 3.1.3): while the nodes of this FSM represent the possible policies, which again are represented by FSMs themselves, the transition function mapping source states and inputs to destination states is formed by the system's meta-policy.

In terms of the self-organizing production cell, the local policy of each agent always adheres to the (hierarchical) finite state machine in Fig. 4.3, which is parameterized by the robot agent's current capabilities and role allocation. An overall system policy is formed by the parallel composition of the robot agents' behaviors together with the carts' behaviors; it is dependent on all the capabilities, role allocations, and cart routes, i.e., the configuration. Though finite in principle, the full system behavior at a given point in time would, due to its sheer size, be rather hard to model in full.

Further, even though we know the input alphabet of the outer adaptation FSM – it is spanned by the possible environmental faults – the concrete transition function formed by the system's meta-policy is assumed unknown at design time (cf. Sect. 4.1.1).

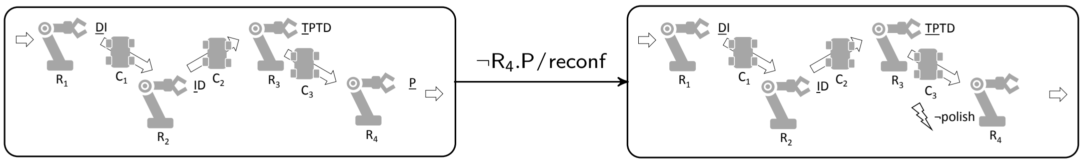


FIGURE 4.5: Exemplary transition between two configurations of the adaptive production cell

By activating an environmental fault in simulation and observing the effect we can, however, infer a transition just as the one sketched in Fig. 4.5 at runtime. Doing so several times with various inputs and making use of a learning algorithm for FSMs (see [BP13] for an overview) we are generally able to infer a single, flat FSM M_{inf} capturing the made insights. Since the system itself is not supposed to learn at runtime (cf. Sect. 4.1.1) the resulting, inferred FSM indeed reduces the number of non-deterministic transitions to be handled in further testing.

4.2.2 Integrating Code-level Logs

For integration of the model-based viewpoint established by the hierarchical FSM with the concrete implementation in terms of the source code we instrument the SuT in the following way: we label each interesting point in the source code with a particular identifier (id) which is output each time it is passed (on top of the originally specified outputs).

Let us, for now, assume that each branch in the source code is considered interesting and thus augmented with an id (a fault-based perspective will be suggested in Sect. 4.3). In terms of the inferred state machine M_{inf} this instrumentation means that its output alphabet is extended with the entire list ID of branch identifiers; and the output function includes a sequence of branch ids.

As concrete implementation for the output ids we proposed so-called *label vectors*, i.e., binary vectors v of length $|ID|$ with $v_i = 1$ if the point in code labeled with ID_i is covered and $v_i = 0$ otherwise.

The exemplary transition within the self-organizing production cell might in consequence be modified as depicted in Fig. 4.6. It is important to understand that (1) equal labels could be output at different transitions, but also (2) there could be labels which are only output at specific transitions. Reaching particular labels by traversing the whole model can be thus viewed as a search potentially involving a targeted use of the adaptation FSM.

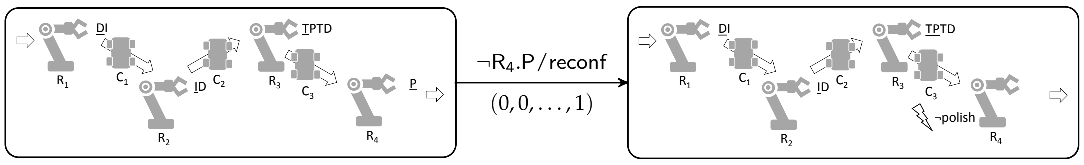


FIGURE 4.6: Exemplary transition between two configurations within the adaptive production cell

The type of model presented offers the special charm of integrating aspects of both, the model-based as well as the white-box testing approaches: we can still apply the classical FSM approaches for systematically fulfilling coverage criteria such as transition or state coverage; but on the other hand, we can also compute paths through the graph of the FSM fulfilling particular white-box requirements based on the output labels.

For this second aspect, the sequence of interesting paths indicated by the output labels can be viewed as kind of meta-branches in the source code (but, in fact, they can include more than one branch in the program). Coherently, the combination of those branches spans some kind of meta-paths within the code (though our representation of label vectors indeed sets limits to path-based criteria since the order in time is not considered). In this way we can define code-based coverage criteria, which in fact are abstract counterparts to the classical code coverage criteria, using the model.

4.3 Mutation-based Labeling

Up to here every branch within the source code was considered labeled. Within our experiments we, however, found that depending on test requirements a further refinement can be valuable. Indeed, a lower number of labels will lead to a lesser number of test cases needed for covering them. In consequence a possibly more precise choice of labels leads to a possibly more efficient set of test cases in future iterations.

In order to show this relationship and to illustrate how such a refinement can be done, we considered a mutation-based test requirement for which an adequate set of labels is to be found.

4.3.1 Mutation-based Test Requirements

Mutation-based test requirements assess the quality of test cases by their ability to uncover slightly modified versions of a SuT [DLS78]. With the intention to simulate possible failure scenarios, these modifications, the so called *mutants*, are generated by well-defined *mutation operators* that mimic typical faults. A test case $tc \in T$, which in terms of the FSM-based model can be considered a particular transition, uncovers

— or more martially *kills* — a mutant if it is able to distinguish the mutant's behavior from that of the original version of the SuT.

A mutant m can be described by a predicate on test cases:

$$\mathit{isKilled}_m \subseteq T \; .$$

If $\mathit{isKilled}_m(tc)$ holds, it is usual to say that mutant m is killed by test case tc. Given mutants M, the killing set of tc is determined by

$$K_{tc} = \{m \in M \mid \mathit{isKilled}_m(tc)\} \; .$$

The confidence that test cases are able to reveal faults in real operation is quantified by a *mutation score*: The greater $|K_{tc}|$, the better the score for test case tc; and the greater $|\bigcup_{tc \in TS} K_{tc}|$, the better the score for a whole test suite TS.

Note that Kapitel 5 will consider a further variant of this traditional mutation score.

4.3.2 Mutation Operators for SO systems

In terms of SO systems successful reorganization requires correct state perception by the agents, correct computation of internal state adaptations by the reorganization mechanism, and correct realization of the delivered adaptation tasks by the agents. Since all of these critical routines are interconnected through message passing between the agents and the reorganization mechanism, a common cause for system failures are errors in communication. For our case study we elaborated the following exemplary mutation operators which mimic those typical reorganization errors:

- *Lost Reconfiguration Message (LRM)*: As soon as a robot detects an environmental fault (e.g., a broken driller) it should normally send a *reconfiguration message* to trigger the reconfiguration mechanism. This mutation operator suppresses such messages. In consequence there might be no reconfiguration in spite of an incorrect system configuration.

- *Needless Reconfiguration Message (NRM)*: The inverse of an LRM: A particular robot signals an environmental fault, although there actually is none. Consequently, unnecessary reconfiguration steps might be triggered.

- *False Reconfiguration (FR)*: This operator mimics the loss of a role allocation message that was sent by the reconfiguration mechanism. In consequence, one robot will (maybe erroneously) retain its previous role — this could result again in an incorrect system configuration.

4.3.3 The Need for Higher Order Mutants

Traditionally, mutation-based test requirements only consider mutants generated by a one-time application of a single mutation operator. Those mutants are called *first-order mutants* (FOMs). *Higher-order mutants* (HOMs) which are generated by multiple applications of mutation operators are usually neglected.

This conforms with the so-called *coupling effect* which states that test cases constructed for explicitly killing FOMs do implicitly also kill combinations of them, as which HOMs can be seen [Off89]. There are, however, known exceptions that suggest to explicitly consider HOMs. Jia and Harman investigate the role of HOMs and give a classification based on the way they are *coupled* and *subsuming* [JH08].

For SO systems in general, and our experimental case in particular, we identified two phenomena in the interrelation of system faults which might lead to unwanted effects if they would only be considered as FOMs, not explicitly combined in HOMs:

(1) *Failure Masking*

Two or more injected system faults might mask the effect of one another. If so, their combination cannot be killed by any test case. Let us consider the exemplary FOMs m_1 and m_2 with separately injected faults f_1 and f_2; and the HOM with both faults combined, denoted with $\{m_1, m_2\}$. If f_1 and f_2 are masking the failure of one another we see that:

$$\forall tc \in T \, . \, isKilled_{m_1}(tc) \wedge isKilled_{m_2}(tc) \rightarrow \neg isKilled_{\{m_1,m_2\}}(tc) \; ,$$

where $isKilled_{\{m_1,m_2\}}(tc)$ for a test case tc evaluates to true if the higher order mutant $\{m_1, m_2\}$ is killed by test case tc. We observed 24 of such cases within the experimental case, which proves the existence of this phenomenon.

Effect: A test case that was originally selected for revealing failures from one of these faults might falsely pass in real operation if the program comprises both.

Instance: One robot erroneously sends a reconfiguration message while another does erroneously not. Considering these faults separately we would either observe that no reconfiguration took place although it should or that a reconfiguration was unnecessarily triggered. In combination, however, a reconfiguration takes place and offers the chance that a correct configuration is chosen.

(2) *Failure in Combination*

Particular failures might only be visible if the program comprises two or more system faults at the same time. If so, there could be test cases not revealing one of the separate faults but doing so for the combination. For the exemplary mutants m_1, m_2 and $\{m_1, m_2\}$ this means that

$$\exists tc \in T \, . \, \neg(isKilled_{m_1}(tc) \vee isKilled_{m_2}(tc)) \wedge isKilled_{\{m_1,m_2\}}(tc) \; .$$

We observed 918 of those cases in our experiment.

Effect: If we exclusively consider FOMs, each of those cases reduces the probability to reveal the HOM in real operation. If this probability goes to zero, i.e., $\neg\exists tc \in T \, . \, isKilled_{m_1}(tc) \vee isKilled_{m_2}(tc)$ even though $\exists tc' \in T \, . \, isKilled_{\{m_1,m_2\}}(tc')$ the reduced test suite will not include a test case that covers the failure anymore.

Instance: Two robots identify environmental faults that need to be signaled. If one of the robots erroneously not sends a reconfiguration message, there might be no failure as the other robot triggers a reconfiguration though. However, if both robots do not send a reconfiguration message the failure occurs.

Algorithm 2 Naïve mutation-based test design

```
Require: $F \equiv$ system faults which can be injected by mutation operators
         $TS \equiv$ the original test suite
$TS' \leftarrow \emptyset$
$Killed \leftarrow \emptyset$
for all $tc \in TS$ do
    $K \leftarrow \emptyset$
    for all $f_1 \in F$ do
        for all $f_2 \in F$ do
            $som \leftarrow$ SOM comprising $f_1$ and $f_2$
            $fom_1, fom_2 \leftarrow$ FOMs comprising $f_1$ and $f_2$, resp.
            for all $m \in \{som, fom_1, fom_2\} \setminus Killed$ do
                if $isKilled_m(tc)$ then
                    $K \leftarrow K \cup \{m\}$
                    $Killed \leftarrow Killed \cup \{m\}$
    if $|K| > 0$ then
        $TS' \leftarrow TS' \cup \{tc\}$
return $TS'$
```

If we want to bypass the mentioned unwanted effects we have to explicitly consider HOMs in test requirements. Yet the question is how to systematically approach such a requirement in test design, as evaluating the mutation score needs test execution — and the more mutants, the more of those executions are needed.

Algorithm 2 shows an exhaustive approach on mutation-based test design for second order mutants (SOMs). It successively selects a number of test cases from a given test suite that cover all possible SOMs. As we can see this approach requires, apart from the effort needed for mutant creation, at worst $|TS| \cdot |FOM|^2$ test case executions in order to investigate $isKilled_m(tc)$. Considering all possible HOMs, this can be generalized to $|TS| \cdot 2^{|FOM|}$.

4.3.4 Application for Labeling

The case study revealed a short-cut that builds on two observations: First, we can observe that only FOMs the injected faults of which are covered by a test case can be actually revealed. This also holds for HOMs: here, each of the injected faults has to be covered. Covering the other lines of code is irrelevant for revealing one of the generated sets of mutants.

Let F be a list of all the possible faults tagged in code. A binary label vector refined for mutation-based testing then has length $|F|$ and indicates which of the faults from F the transition would generally cover. Note that in our notation this not necessarily holds for combinations of faults. The one fault might cause changes in control flow so that the other would not be reached anymore. For instance just consider an if-else statement with one fault in the condition and one in a branch. As soon as the fault in the condition is active the taken branch for the same input could change in a way that the second fault is not reached anymore. It follows that if we wanted to cover all the possible dependent faults, we would need to examine all possible paths in the presence of all combinations of faults. In other words, we would still have to simulate the effect of each of the mutants.

This brings us to the second observation we made: the considered mutation operators *LRM*, *NRM*, and *FR* generate system faults which are *independent* from one another, as they are distributed over different entities that are only connected through message passing. A fault in the source code of robot R_1 does not influence the path passed through control flow in robot R_2. In this particular case we know that the path in code drawn by a particular label vector will not change because of active faults. The requirement of covering all HOMs can in consequence be reduced to covering all existing label combinations which can be viewed as a more abstract counterpart to branch coverage in code.

4.4 Representative Subset Selection through Clustering

Let us now consider how to learn a test policy by making use of the proposed kind of model. For the case study we supposed the following setting: Given an SO system such as the self-organizing production cell that is described in Sect. 4.1.2 we applied the simulation-based testing technique described in Sect. 4.1.4 for exploration. By instrumentation, we logged the effect of each of the executed test cases in terms of normal outputs and label vectors (cf. Sect. 4.3) in a file. As a result we were able to construct an M_{inf} as described in Sect. 4.2.1.

We assumed to have given a reset operator which lets us establish the SuT's source configuration at any time. Consequently a test policy was considered to be an enumeration of given test case sequences, all of which starting at the source. The order in which the chosen test case sequences are applied, in this case, does not matter. Our very goal was hence to find a possibly small subset of all possible sequences through M_{inf} that covers all the label vector manifestations contained in the model.

As exhaustive solutions are NP-complete, we investigated the suitability of existing relaxations by linking our task with the so-called *representative subset selection* problem [ESS16; FD07] from the machine learning literature. The task definition there is "finding a subset of a large number of models or data points, which preserves the characteristics of the entire set" [ESS16]. As entire set we considered a test suite *TS* comprising a number of paths through M_{inf} that we generated by random walks. The search we performed for a $TS' \subset TS$ could thus be viewed as kind of test suite reduction.

4.4.1 Finding Representative Subsets using Clustering Techniques

The problem of finding a representative subset from a huge number of models has been extensively studied in the machine learning literature. A lot of algorithms, including those of Elhamifar, Sapiro, and Vidal [ESV12], Frey and Dueck [FD07], Kaufman and Rousseeuw [KR87], Boutsidis, Mahoney, and Drineas [BMD09], Tropp [Tro09], and Balzano, Nowak, and Bajwa [BNB10], were proposed to tackle this problem in different facets. Common use cases include recommender systems, computer vision, or language processing. To the best of my knowledge, there have been no investigations yet applying these algorithms to test suite reduction or test design.

A prominent group of algorithms tackling the representative subset selection problem operates on measurement vectors for data that lies in low-dimensional subspaces [BMD09; Tro09; BNB10]. Its members are instantiated only by the measurement vectors describing the data. Since we for the case study, however, assumed that label vectors as we consider them here should not be interpreted as arbitrary measures in a vector space and that a particular dissimilarity metric is needed, we concentrated on another group: algorithms selecting the subset based on pairwise similarities or dissimilarities between the data points of the original set. An adequate measure for test cases will be discussed in the next section. In concrete, our experiments were based on the *Affinity Propagation* [FD07] and the *Dissimilarity-based Sparse Subset Selection* algorithm [ESV12].

Affinity Propagation (AP). This method approaches the representative subset selection task by a message passing algorithm. Each data point is viewed as a node in a network that transmits messages along the edges updated by particular rules. By this process of sending and updating messages an energy function evolves that guides the choice for particular *exemplars*, as the authors call the elements within the representative subset.

The concrete number of output exemplars is determined during the process, which means that we cannot precisely specify the size of the representative subset in advance. By appropriately choosing the diagonal similarities *pref*, i.e., the suitability of points for representing themselves as exemplars, we can, however, approximately control the scale [FD07]. Values between the minimum (few exemplars) and the maximum (many exemplars) possible similarity are common. The algorithm terminates if the cluster boundaries remain unchanged over a number *convIter* of iterations or if a threshold *maxIter* of iterations is reached.

Dissimilarity-based Sparse Subset Selection (DS3). Elhamifar et al. formulate the representative subset selection as what they call a *Row-Sparsity Regularized Trace Minimization Problem* [ESV12] on two matrices D and Z. While matrix D arranges the dissimilarities between the different data points, Z denotes the probabilities, that one of the data points is represented by another.

Based on those matrices an optimization program is formulated that is combining two goals: First, all data points should be represented possibly well, meaning the the dissimilarity between a represented data point and its representative should be possible small; and second, there should be as few representatives as possible.

They propose the DS3 algorithm for solution. In [ESS16] an implementation is presented that builds on the *Alternating Direction Method of Multipliers (ADMM)*. This implementation was used for our experiments.

Like AP, also DS3 is not directly parametrized by the concrete number of representatives. Instead, a regularization parameter ρ can be set which controls the scale. A second parameter *maxIter* determines the maximum number of iterations. If an internally computed error does not fall below a threshold ϵ in $k <$ *maxIter* steps, the algorithm terminates.

4.4.2 A Dissimilarity Measure for Test Sequences

As stated before, we claimed that for a reasonable clustering of transition sequences by the included label vectors w.r.t. our goal of covering as many of the mutants as possible a specific dissimilarity measure is needed. Consider for instance two particular transitions t_1 and t_2 with label vectors $t_1.v$ and $t_2.v$ in their outputs (we access a label vector v of transition t by $t.v$); and assume we would represent t_2 by t_1 which means that we would exclusively test transition t_1. As uncovered mutants cannot be killed, the number of mutants we could possibly miss by not testing t_2 may be estimated by $|\{l \in \{1, \ldots, n\} \mid t_2.v_l = 1 \wedge t_1.v_l = 0\}|$, where n is the number of introduced labels, i.e. $|F|$.

Note that in addition to the missed FOM, we could possibly miss some HOM that would only be killable in combination with a non-covered FOM (cf. pattern (1) in Sect. 4.3.3). Contrary, each mutant exclusively covered by t_1, but not by t_2, could cause failure masking issues (cf. pattern (2) in Sect. 4.3.3) and thus increase the chance of missing HOMs again. This can be estimated by $|\{l \in \{1, \ldots, n\} \mid t_1.v_l = 1 \wedge t_2.v_l = 0\}|$.

These considerations suggest the following dissimilarity measure for single transitions based on their label vectors, which can be interpreted as comparing two bit patterns by the *Hamming distance* [Ham50]:

$$Dist(t_1, t_2) = |\{l \in \{1, \ldots, n\} \mid t_1.v_l \neq t_2.v_l\}| \,.$$

For quantifying the dissimilarity between whole transition sequences ts_1 and ts_2 the preceded metric is lifted by summing up the minimal distances between the test cases from the one path to those of the other.:

$$Dist(ts_1, ts_2) = \sum_{t_2 \in ts_2} \min_{t_1 \in ts_1} Dist(t_1, t_2) \,. \tag{4.1}$$

This can be transformed to a similarity measure by

$$Sim(ts_1, ts_2) = -Dist(ts_1, ts_2) \,. \tag{4.2}$$

Note that (4.1) as well as (4.2) are not symmetric and hence violate the properties of metrics. They quantify, how well and how worse ts_2 is represented by ts_1. Both considered algorithms for representative subset selection can cope with this aspect.

Using the dissimilarity/similarity measure for transition sequences and an enumerable test suite TS, we can generally construct a $|TS| \times |TS|$ similarity matrix S as well as a dissimilarity matrix D with the same dimensions by $S_{i,j} = Sim(TS_i, TS_j)$ and $D_{i,j} = Dist(TS_i, TS_j)$ for all $0 \leq i, j \leq |TS| - 1$. These matrices served as inputs for the subset selection algorithms.

4.4.3 Application on Case Study

For evaluation we applied the discussed approach to the experimental case described in Sect. 4.1. Because of the huge number of possible transition sequences within the learned model, which in fact is unfeasible for the subset selection algorithms (at least for our computing capacity), we did not take into account all of them but based our experiments on a test suite TS to reduce comprising 500 sequences generated by random walks.

Using the distance function in (4.1) and the mutation operators *LRM*, *NRM* and *FR* (cf. Sect. 4.3) the matrices S and D were constructed. For assessing the degree of optimization w.r.t. the given similarity/dissimilarity notion an informal search found the following empirically derived parametrization being suitable:

- *AP*:
  - *pref*: equally drawn from $[m-2, m+2]$ where m is the median of similarities in S
  - *convIter*: 15
  - *maxIter*: 3000
- *DS3*:
  - ρ: equally drawn from $[0.001, 0.005]$
  - *maxIter*: 3000
  - ϵ: 10^{-7}

We additionally applied a uniform sampling of k transition sequences for each k proposed by the AP or DS3 algorithms serving as a baseline. Building on the previous considerations we evaluated the reduced test suites $TS' \subset TS$ by a *representation cost measure*:

$$\sum_{ts \in TS} \min_{ts' \in TS'} Dist(ts, ts') \ .$$

The idea behind was to estimate the number of mutants covered by the one suite but not by the other by comparing and summing up the distances of the respectively most similar sequences.

The left plot in Figure 4.7 presents the results for the AP, the DS3 and the uniform samples on our case study. As we can see, both classifiers are more or less on the same level; they outperform the uniform samples regarding the representation cost. The right graphic, which depicts the actual number of killed mutants (FOMs and SOMs), suggests that optimizing the representation cost also optimizes the number of actually killed mutants.

We see that test suites reduced by the classifiers kill more mutants than the uniform samples for the same k. Even if not all of the killable mutants (6245 in our experiment) are actually killed, the DS3 algorithm got 91.3 % of them with 62 paths including 247 test cases which is about 3 % of the test cases of the original suite.

4.5 Related Work

The experiments reported in this chapter reduced test policy learning to a test suite reduction task. Yoo and Haman characterize this problem, which is traditionally considered as one of the major challenges for regression testing, as finding a representative subset of test cases T that still satisfies all imposed test requirements

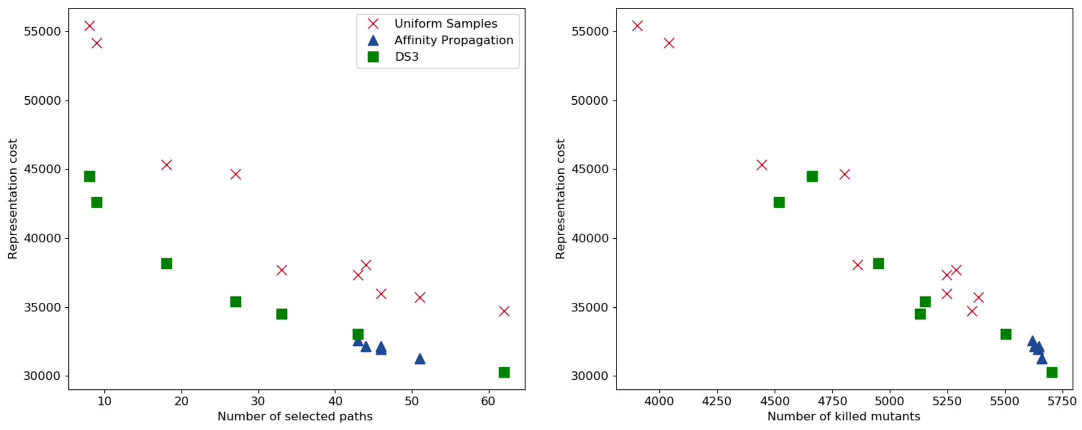


FIGURE 4.7: Results of the Mutation-based Test Suite Reduction using the classifiers.

[YH12]. According to the authors this can be considered as the *minimal hitting set problem*, which is in general NP complete. For the purpose of efficiency, methods approaching test suite reduction typically involve heuristics. Many of them, such as the GE and GRE heuristics [CL96], can thereby be viewed as variations of the greedy approach on the set cover problem proposed by Papadimitriou and Steiglitz [YH12; PS98].

More general, existing research on test suite reduction can be classified by (1) the type of coverage criterion and (2) the quality measure they use for reduced test suites [Shi+14]. While it seems quite common to use fault-based techniques for (2), for example [Rot+02; BMK04; Zha+11; FW07; Rot+01], for (1) the huge majority of approaches rather applies code coverage criteria, such as statement, branch or decision coverage, cf. [BMK04; Zha+11; JH03; Rot+02].

I found two exceptions that also based (1) on mutants, similar as we did: Offutt, Pan, and Voas propose a procedure for mutation-based test suite reduction called *ping-pong* [OPV95]. This applies elaborated heuristics in order to efficiently find a subset of test cases that kills all the mutants that were killed by the original set, too. They show, that they can reduce mutation-based test sets by over 30%. In contrast to us they exclusively consider FOMs. As they show, the introduced heuristics reduce the number of test cases to execute before reduction. Still, during the search for the reduced set, each chosen test case has to be executed one time per mutant to kill. If they could use the label vectors we proposed, they could apply a similar short cut as we did (see Sect. 4.3.4). However, they actually cannot, as the faults they consider are not independent from one another.

Shi et al. evaluate classical test suite reduction algorithms on requirements called *Mutant Adequate Reduction* (MAR) and *Statement-Mutant Adequate Reduction* (SMAR) [Shi+14]. While the first exclusively considers killed mutants the second combines this with code coverage. They use the mutation tool PIT for evaluation on open-source projects. Again, exclusively FOMs are considered and again, label vectors cannot be used for reducing the number of test cases to execute before reduction.

To the best of my knowledge no study considers test suite reduction for the particular system class of self-organizing systems. [Ebe+17c] addressed the related task of test case selection. They present a couple of strategies which leverage particular characteristics of self-organizing systems and test oracles in order to speed up a search-based approach on test input generation. In contrast to that, our case study was concerned with reducing an already existing set of test cases whose elements are being executed at least once.

4.6 Conclusion

This chapter reported about a case study of simulation-based test design for a self-adaptive system which is not learning at runtime. It showed that in case the system's meta-policy is fixed we can infer a test model for further test design from simulation.

More specifically, the case study utilized a mutation-based approach on prioritizing or, as we understood the test policy learning task, reducing a given set of test case sequences. The approach combined model-based and code-based test design techniques: we inferred a finite state model from simulation where the transitions were enriched by label vectors for recording mutations in the code . Thereby we took first-order as well as higher-order mutants into account that simulated the effect of communication faults during the reconfiguration process. The use of established clustering techniques for representative subset selection showed promising results w.r.t. our experimental case.

Regarding the particular approach proposed, possible directions of future work include the analysis of the impact of outliers on the overall representation cost by a reduced test suite and a more detailed investigation of adequate mutation operators for self-organizing systems.

In the following chapter I want to address, however, a more general issue: can we, instead of just reducing sets of already executed test case sequences, also use the gained insights to shape fully new ones that are possibly profitable in terms of a given goal? If so, we could establish a kind of feedback loop in simulation that explores and evolves the wanted test policy for particular test goals.

Chapter 5

Test Policy Evolution

The latest chapter's experiments considered devising test policies for testing self-organizing systems. Given a simulation of a particular self-organizing system's environment a behavioral model of this system had been derived from simulation once which could be used then to base test policy selection for further testing on this. We regarded test policy learning as a reduction task and tried to find a subset of all the test case sequences we applied in simulation, such that the chosen sequences were still representative for the rest. This representativity-based test goal showed to trigger good results w.r.t. mutation-based testing: The more representative a test sequence for the rest, the more mutants were killed.

Still, it is to consider that due to limits in cost and time we will usually not be able to execute test suites that are likely to kill all theoretically killable mutants. We should thus be aware of the possibility that even after successful execution of the test suites there might be unrevealed faults existing which were actually anticipated by means of a mutation operator but still are leading to worst failures in real operation.

This shows a natural disadvantage of mutation-based test goals: since only the count but not the kind of triggered failures of killed mutants is taken into account mutants standing for irrelevant damage count exactly as much as those standing most enormous ones. As we are not always able to measure the effect of faults before executing the faulty program in real operation, we will have to live with this weakness in general. In case of self-adaptive systems with given simulated environments (see Sect. 3.2) we, however, can measure the effects using the simulation.

Present chapter reports about experiments we performed about modifying the mutation-based test goal in a way that it is taking into account not only the sheer number, but also the assumed effect of revealed mutants. This guides us to a novel, *weighted* mutation score (Sect. 5.1.2). For designing test suites that are meeting this goal the eligibility of meta-heuristic search approaches, or, more concretely, of custom variants of classical evolutionary algorithms is studied.

By the reported experiments we presented a novel evolutionary mutation as well as a recombination operator that are particularly suitable for solving sequential optimization problems (Sect. 5.2), but may also be useful for various applications beyond test design. An evaluation shows promising results (Sect. 5.2.4). Encouraged by these results Sect. 5.3 elaborates a possible path for future work before this chapter is concluded by related (Sect. 5.4) and future (Sect. 5.5) work.

The reader should note that some parts of this chapter are direct citations from papers [RGK18] and [Rei+16] which are set in context of present work.

Algorithm 3 Mutation-based test suite evaluation

```
Require: $p \equiv$ reference version of the program under test
         $O \equiv$ set of mutation operators
         $S \equiv$ mutation score function
function killed($ts$)                                 ▷ $ts$: test case input sequence of length $|ts|$
    $K_{ts} \leftarrow \emptyset$                     ▷ map of killed mutants, indexed by mutants
    for all $o \in O$ do
        $m \leftarrow$ mutated version of $p$ by application of $o$
        reset system state
        for $t \leftarrow 1..|ts|$ do                 ▷ iterate through time steps
            $eff \leftarrow p.execute(ts(t))$
            $eff_m \leftarrow m.execute(ts(t))$
            if $eff \neq eff_m$ then
                $K_{ts}[m] = (eff, eff_m)$            ▷ mutant killed
                break                                 ▷ continue with next mutation operator
    return $K_{ts}$                                   ▷ return killed mutants
function $\Gamma_{\mathrm{M}}(TS)$                    ▷ $TS$: test suite of test input sequences
    $K \leftarrow \emptyset$                          ▷ map of killed mutants, indexed by test input sequences
    for $ts \in TS$ do                                ▷ iterate through test input sequences
        $K[ts] \leftarrow$ killed$(ts)$
    return $S(K)$                                     ▷ return mutation score
```

5.1 From Mutation to Risk-based Testing

The very goal in mutation-based test design is finding a test suite whose test cases are supposed to kill as many mutants as possible. Recall that mutants are generated from so-called mutation operators O. Those are a applied to a reference version p of the program to test and simulate the effect of introduced errors.

Each of the mutants is executed with each of the test cases from a test suite and those mutants that show an effect deviating from the effect of p are recorded; such a mutant is said to be killed by the test case. Finally, for evaluating a test suite, a *mutation score function* S is applied to the record of killed mutants. See function $\Gamma_{\mathrm{M}}(TS)$ in algorithm Alg. 3 as outline of a test goal procedure evaluating a test suite TS comprising a set of test case sequences on this way.

In the normal case that was also considered in Kapitel 4 the score function merely counts the number of killed mutants, i.e.:

$$S_c(K) = |\{m \mid K[m] \neq \emptyset\}| \,. \tag{5.1}$$

No prioritization between the single mutants is established which means that the killing of all mutants is supposed to be of equal value for the test suite. In other words, if we could choose for a subset of fixed size of all mutants to be revealed by the suite, we could just pick random ones and obtain the maximum score.

However, the possible effects of faults mocked by the mutants, i.e., the severity of resulting failures, diverge. While particular failures might cause whole production stops and thus appear very costly, others might be easy to fix and do not really matter.

As premise of experiments presented in the following we assumed that it would be hence more reasonable to prioritize potentially harmful faults higher than supposed harmless ones in order to fix them early and to avoid the most hazardous failures. This is also backed by literature about *risk-based testing* as Sect. 5.1.1 shows by some exemplary testing approaches bringing a viewpoint of risk into test design. Section 5.1.2 shows how this risk-based viewpoint can be combined with mutation-based testing by means of a *weighted* mutation function.

5.1.1 Risk-based Testing

Building on general high-level considerations from authors such as Bach [Bac99] or Amland [Aml00], several methods for integrating *risk estimations* in software testing evolved though at different levels of automation. Many of them again build on test models of the SuT which are getting annotated by risk values for formulating risk-based test objectives and figuring out the test design:

Kloos et al. [KHE11] derive such risk annotations from the results of a fault tree analysis from which test cases can be generated. Bauer et al. [Bau+08] transfer the risk of annotated UML diagrams to a test model, from which test cases are derived. Zimmermann et al. [Zim+09] extend this approach by refining the test models so that from these only so-called *critical* test cases are generated. Wendland et al. [WKS12] propose to formulate requirements for the SuT in so-called *integrated behavior trees*. These are annotated with risk values associated with certain risk levels. A risk-optimized test suite is generated from the annotated models by using test directives. In all of these approaches the risk assessment is done by experts.

Stallbaum and Metzger [SM07] note that the *risk assessment* of test cases done by experts could get a critical cost factor. They propose an approach that automates the risk assessment based on requirement metrics. Such metrics refer for example to the revision frequency or the cyclomatic complexity of a use case. The use of similar metrics was also proposed by Amland [Aml00]. He calculates so called *risk indicators* for every function of the SuT from which the occurrence probability of failures can be estimated. The exposure of possible failures is quantified by expert estimates.

Since all of the mentioned approaches rely on deterministic (or at least probabilistic) models of the SuT's behavior, they are not directly applicable for testing self-adaptive systems with their non-deterministic behavior. And in fact, it is certainly not possible to assess the effect of particular faults if we actually do not know which approach the system will actually take at runtime. Using experience from previous test runs to derive the actual behavior from observation as discussed in previous chapters, we however get a chance. As the experiments will show (see Sect. 5.2.4), given a simulation of the self-adaptive SuT and its environment it is indeed possible to evolve a test suite for meeting risk-aware goals.

5.1.2 A Weighted Mutation Score

The aim of reported experiments was to integrate viewpoints from risk-based testing into the mutation-based testing methodology as it was described in the previous section. Assuming that occurrence probabilities of faults might be easily added afterwards in operational settings (they might be obtained from historical test results) the term risk, which normally includes the occurrence probability as well as a quantification of an events exposure, was thereby firstly reduced to its exposure factor: we suggested to weight the traditional mutation score by a measure of failure severity. Instead of counting all the mutants killed, the weighted mutation score sums up the severity of those.

Experimental Setting For the experiments we reused the case study described in Sect. 4.1.2 and considered the issue of testing a self-organizing production cell. Recall that according to our concept of a self-organizing system this production cell comprises multiple agents that are coordinated by a *reconfiguration mechanism* (see Sect. 4.1.1). In more detail, the production cell comprises robots and carts whose configuration is given by roles for the robots defining their concrete behavior and routes for the carts connecting the robots (see Fig. 4.2).

Just as described in Sect. 4.1.3 the production cell's wanted behavior is specified by means of the *Restore Invariant Approach* (see for example Fig. 5.1): Instead of specifying the concrete system approach at runtime only a set of predicates on the environment is given sketching the system's objectives by what we called the *Corridor of Correct Behavior (CCB).*

If one of those predicates, say that each task for the production cell can be processed and completed with the available resources, evaluates to *false* a so-called reconfiguration mechanism is triggered to compute and distribute a valid configuration again. To test this behavior, we are able to confront the system with so-called *environmental faults*, such as broken drillers of robots or obstructed routes for carts. Activating a particular environmental fault thereby serves as test input.

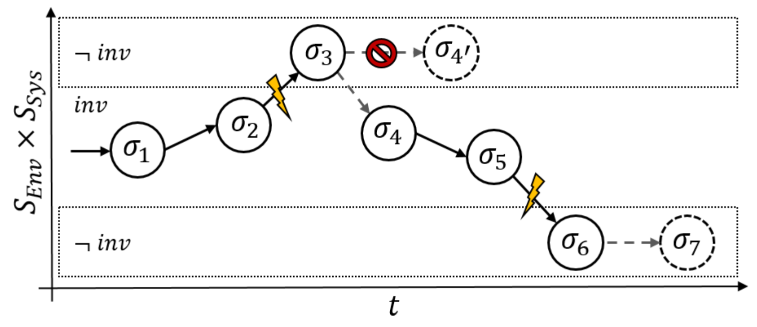


FIGURE 5.1: The overall system state changes at runtime due to environmental dynamics. In case the invariant is getting broken – say because some unforeseen disturbance occurred – the system is meant to trigger reorganization in order to establish a configuration that, in combination with the environmental state, is located inside the corridor again. If no such configuration is eventually established even though it would be generally possible, a failure within the integrated implementations occurred (see $\sigma_{4'}$).

Severity Levels As a first step towards the weighting of failures and building on the CCB (cf. Fig. 5.1) we proposed to classify test results (as we call the effect observed after executing a program with a test case) by different severity levels: If test execution results in reorganization, i.e., the reorganization mechanism transferred a state outside the corridor to the inside again, we assign the result to the class *reorg*. Otherwise, we assign it to $\neg reorg$. On this way function $C : \mathit{Eff} \rightarrow \{reorg, \neg reorg\}$ determines the class of an effect $\mathit{eff} \in \mathit{Eff}$.

Comparing the effects eff and eff_m as they are gained in line 7 and line 8 in Alg. 3, we quantify the severity levels of the four possible permutations with a severity function $Sev : \mathit{Eff} \times \mathit{Eff} \rightarrow \mathbb{R}$:

$$Sev(\mathit{eff}, \mathit{eff}_m) = \begin{cases} 1 & \text{if } C(\mathit{eff}) = C(\mathit{eff}_m) \\ 2 & \text{if } C(\mathit{eff}) = reorg \wedge C(\mathit{eff}_m) = \neg reorg \\ 3 & \text{if } C(\mathit{eff}) = \neg reorg \wedge C(\mathit{eff}_m) = reorg \end{cases} \tag{5.2}$$

The first case is obviously the most harmless one. If a test case triggers reorganization in both program versions (or in both versions not) one can argue that no real failure was detected. However, as the mutant is killed though (cf. line 10 in Alg. 3) we assign a slight severity score to this case. The second as well as the third case indicate that the killed mutant simulated a real failure. In the second case, no valid state was established even though this would be possible. Such a failure would require human intervention in real operation.

The third case, in which a valid state was established even though this is generally not possible, implies even higher costs in real operation, as it mostly results in a contradiction between software and hardware. Those quantified severity levels are taken as basis for the weighted mutation score

$$S_{\mathrm{w}}(K) = \textstyle\sum_{\{m:K[m]\neq\emptyset\}} Agg(Sev(K[m]))\ , \tag{5.3}$$

where Agg aggregates the various severities observed when a single mutant is killed by more than one test case from TS.

In experiments we thereby proposed to instantiate the operator Agg with $\sum$ or max and suggested to use $\sum$ if the errors simulated by the mutation operator are assumed to be *transient*, which means that the error does not always trigger a failure if covered. For the others, the *persistent* errors, we suggested to use max.

5.1.3 Test Case Dependency Graph

Given the novel test goal described the challenge is finding a test suite TS that optimizes this goal in terms of the obtained score. In experiments we further demanded that $|TS|$ conforms with a predefined maximum number of investable time steps k such that if each test sequence of TS has a length of (at most) $|ts|$, $k = |TS| \cdot |ts|$.

Recall that in case of a self-adaptive, such as a self-organizing, SuT the search for such a test suite brings the following two issues:

1. The effects of test cases in terms of killed mutants are dependent on the full history of previously executed test cases in a test sequence due to reconfigurations as

$$\begin{tikzcd}
 & & \text{initial} \arrow[dl, "tc_1"'] \arrow[dr, "tc_2"] & & \\
 & \mathit{eff}_1\{m_0[2], m_2[1]\} & & \mathit{eff}_2\{m_4[2]\} \arrow[dl, "tc_3"'] \arrow[dr, "tc_7"] & \\
 & & \mathit{eff}_3\{m_2[3], m_5[1]\} \arrow[dl, "tc_4"'] \arrow[d, "tc_5"] \arrow[dr, "tc_6"] & & \mathit{eff}_7\{m_2[1], m_6[2], m_7[3]\} \\
 & \mathit{eff}_4\{m_1[1]\} & \mathit{eff}_5\{m_3[1], m_6[1], m_8[1]\} & \mathit{eff}_6\{m_9[3], m_7[1]\} &
\end{tikzcd}$$

FIGURE 5.2: Exemplary dependency graph with seven test cases $tc_1, \ldots, tc_7$ annotated by the set of killed mutants respectively. The numbers in squared brackets denote the severity of killed mutants. The best test suite of size $|TS| = 2$ for max-aggregation comprises of the two sequences $ts_4 = \langle tc_2, tc_3, tc_6 \rangle$ and $ts_5 = \langle tc_2, tc_7 \rangle$ scoring 14. For $\sum$-aggregation, however, the best test suite with $|TS| = 2$ is $\{ts_3, ts_5\}$ with $ts_3 = \langle tc_2, tc_3, tc_5 \rangle$ with a score of 19.

adaptations to these previous environmental influences; in particular, executing a test case influences the future scores.

This was previously formalized by hierarchical state machines describing the actual system behavior (Sect. 4.2). Due to the limit k of investable time steps, here we rather consider traces with maximum length through those machines and said that the search space for the optimal test suite is given by a *dependency graph* with the initial system state as root, effects and their killed mutants as nodes, and the test cases as edges; see Fig. 5.2 for a small example.

2. Each evaluation of a test suite TS is at the cost of $k \cdot |O|$ program executions at worst. The only factor that we can influence for practicability is thus the number of evaluations that has to be kept to a minimum.

Note that in context of the dependency graph test design can be seen as a general optimization problem: find a number of paths through a graph in the most efficient way, such that their collected nodes optimize a given goal. For the aggregation by $\sum$, when disregarding that test suites are sets, a single best path could just be repeated, and optimization would be reduced to the well-established problem of finding a single path with maximum score [McM04]. The aggregation operator max, however, directly considers sets of nodes for evaluation and is sensitive to duplicates; greedy approaches iteratively choosing the single best rated path are doomed to fail.

5.2 Evolutionary Algorithms

The machine learning techniques that were used in Kapitel 4 for test design could not be applied anymore since they were exclusively designed for finding representatives – a test suite that is possibly representative for the whole set of all the test cases, however, does not necessarily have to cover the optimum set of mutants w.r.t. their weights. In other words, there is no functional dependency between the label vectors spanning the representativity and the properties determining the weight.

Thus, with evolutionary algorithms the experiments were based on another technique for optimization. Present section introduces this kind of algorithms Sect. 5.2.1 and shows how we customized it for the specific needs of deriving test case sequences at test design Sects. 5.2.2 und 5.2.3.

5.2.1 Foundations

Evolutionary algorithms are a wide-spread optimization technique that does not require a gradient on the solution space to be computable [ES03]. Any evolutionary algorithm works on a set of solution candidates, also called *individuals*, a set of which is also called a *population* P. For our experiments we defined a single individual TS to be a whole test suite. The set of currently considered test suites thus formed P as a subset of the domain of all possible test suites.

Given those definitions Alg. 4 shows the typical structure of evolutionary algorithms. It starts with a random initialization and repeats its other operations for a fixed amount of times n. Each of these repetitions is also called a *generation*.

Random Initialization. This step generates the initial population by generating random test suites. Note that *generate* is not a mathematical function as it returns a newly generated object each time it is called. Here, the term *genetic operator* is used for common evolutionary operations that use random effects.

Recombination. We decided for a variant of recombination that grants the chance to recombine to each individual (irregardless of its fitness), but chooses its respective mate with respect to higher fitness. Effectively, this seems a good compromise between allowing exploration (using all individuals for recombination) and exploitation (favoring the better ones). The former is guaranteed by applying a fixed chance r_{recomb} for the choice of any individual for recombination. The randomized function *select_mate* performs the latter by iterating over the population, returning the nth-fittest individual with probability 2^{-n}.

We first create an empty test suite (i.e., containing no test sequences but already of required size $|TS|$) in the variable *child* and iterate over the number of test suites that is used for all our suites and complete the child by performing one random choice of three operations with equal probability (as denoted by the **or** operator): (a) we reuse the test sequence of the first (randomly chosen) parent, or (b) we use the test sequence of the second (chosen according to fitness) parent, or (c) we call a special function *combine* that builds a new test sequence out of the test sequences stemming from both parents. Section 5.2.3 will show how to effectively implement such a function. Leaving out option (c) entirely would result in a more standard evolutionary algorithm that still manages to produce effective (but not as good) test suites (see Sect. 5.2.4). The recombination can then be considered as a standard uniform crossover at the whole suite level.

Note that even though recombination is a common step that is an integral component of almost all evolutionary algorithms, we did not present it as a black-box

Algorithm 4 Evolutionary Algorithm for Test Suite Generation

```
Require: $n \equiv$ maximum amount of generations
         $m \equiv$ maximum amount of individuals in the population
         $r_{\text{recomb}}, r_{\text{mut}}, r_{\text{hyper}} \equiv$ rates of evolutionary operators
         $evaluate \equiv$ fitness/objective function
         $rnd \equiv$ random number generator on codomain $[0, 1]$
         $generate \equiv$ genetic operator that randomly generates a test suite
         $mutate \equiv$ genetic operator applying small changes to a test suite
         $combine \equiv$ function combining two test sequences to produce a new one
         $select\_parent \equiv$ function returning a mating candidate in a population
$P \leftarrow \emptyset$
for $j = 0, \ldots, m-1$ do                                    ▷ Random Initialization
    $P \leftarrow P \cup \{generate()\}$
for $i = 0, \ldots, n-1$ do
    for all $TS \in P$ do                                      ▷ Recombination
        if $rnd() < r_{\text{recomb}}$ then
            $mate \leftarrow select\_parent(P)$
            $child \leftarrow (\textbf{null})^{|TS|}$
            for $k = 0, \ldots, |TS| - 1$ do
                $child[k] \leftarrow TS[k]$ or $mate[k]$ or $combine(TS[k], mate[k])$
            $P \leftarrow P \cup \{child\}$
    for all $TS \in P$ do                                      ▷ Mutation
        if $rnd() < r_{\text{mut}}$ then
            $P \leftarrow P \cup \{mutate(TS)\}$
    for all $TS \in P$ do                                      ▷ Hypermutation
        if $rnd() < r_{\text{hyper}}$ then
            $P \leftarrow P \cup \{generate()\}$
    while $|P| > m$ do                                         ▷ Selection
        $P \leftarrow P \setminus \{\arg\min_{TS \in P} evaluate(TS)\}$
return $\arg\max_{TS \in P} evaluate(TS)$                      ▷ Result
```

genetic operator but put a bit of its implementation into the description in Alg. 4 in order to accurately describe how our implementation of *select_mate* fits in. The function *combine* thus does not accurately represent the whole genetic operation "recombination" the way *mutate* and *generate* do.

Mutation. Each individual is subject to mutation with a chance of r_{mut}. When chosen, the *mutate* operator generates a new individual through small random changes to the original. It is not obvious how a *small* change can be accurately quantified or guaranteed in the domain of test suites. It is, however, important that mutation operates on a small scale as it is our main exploratory operator and large mutations may (systematically) jump over some solutions. This problem is addressed in Sect. 5.2.3.

An alternative to caring about the "smallness" of the changes is to just pick a random test sequence of the suite and re-generate it through random walk within the dependency graph starting at a randomly chosen point in the test sequence, which results in a rather big change with each mutation. These approaches are compared in Sect. 5.2.4. Note that in contrast to some evolutionary algorithms (and biological evolution), mutated individuals are only added to the population instead of having them replace their original counterparts.

Hypermutation. During the hypermutation step, we simply generate new individuals completely at random disregarding the previous course of evolution, and add them to the population. For this purpose, we use the same *generate* operator as in the initialization of the first population. Adding these new individuals increases exploratory behavior of the evolutionary algorithm and thus helps prevent getting stuck in local optima. Note that in parallel to the other operators (and their respective application rates), the amount of generated individuals is based on the number of individuals within the population $|P|$ and the given parameter r_{hyper}. Also note that the phases extension which will be discussed in Sect. 5.2.2 turns the evolutionary algorithm into a dynamic optimization problem, for which the use of hypermutation had been highly suggested [Gre92].

Selection. The selection step is rather straightforward: it simply chooses the m best individuals to keep for the next generation. The description in Alg. 4 uses a notation that does not need to introduce list slicing, although the implementation uses a computationally more efficient functional equivalent to the algorithm presented here.

Result. Finally, the best individual found in the last population of the last generation is returned. However, this is also the best individual that has been found overall – which is ensured by the fact that all of our operations in the steps within each generation only add new individuals but never overwrite their parents. The fitness function *evaluate* applies Γ_{M} to all test cases of the sequences in TS. As long as we do not change its semantics this means that we always keep the best individuals around. This feature is called *elitism* within the field of evolutionary algorithms. While the search process is (even without elitism) expected to strive for better individuals anyway, elitism ensures that it is monotone, as Sect. 5.2.1 will show.

The experienced reader might note that, as is usual when deploying an evolutionary algorithm, a fixed limit $|P| = m$ is set on the population size. Furthermore, we employ a fixed limit of execution time (measured in evaluations or generations as we discuss later) instead of a quality threshold as would be possible as well as break condition. However, especially for our experiments we were most interested in the comparison of the quality of various approaches within a given time frame, as for software testing the requirement is more likely formulated to produce the best test suite within the available time rather than to produce a test suite as fast as possible.

5.2.2 Phases Extension

Having discussed the basic functionality of the evolutionary algorithm we utilized for test suite generation, let me now introduce the first of two extensions we found for the test domain.

As first extension we introduced the so-called *phases extension* improving the overall search performance. As Sect. 5.2.4 will show it manages to produce comparable results with roughly half the goal evaluations.

Generally speaking, we observed that there is a noticeable relation between the fitness of a test suite TS and the fitness of a single test sequence $ts \in TS$, i.e., the fitness of the suite $\{ts\}$. As discussed, the best test suite of size k will usually not consist of the k best rated test sequences, as these will likely overlap in killed mutants and thus have poor overall coverage. However, it seems intuitive to start with one of the best rated test sequences and then build a suite around it. We could thus split the test suite generation problem into various sub-problems of iteratively finding test sequences given certain constraints (from previously found test sequences). But evolutionary algorithms provide us with a much more elegant approach, which we call *phase-based evolution*:

We adjust the objective of the evolutionary process and the data structure of its individuals during the progression of evolutionary search. The evolutionary process is started with individuals that contain test suites $TS \in P$ of size $|TS| = 1$, i.e., all test suites only containing a single test sequence. This evolutionary search runs for the best single test sequence for a while: If we eventually want to search for a test suite TS of size $|TS| = k$ after n generations, we run this reduced search problem for roughly $\frac{n}{k}$ generations. Then we augment all individuals to represent a test suite with two test sequences by adding a randomly generated test sequence to each individual. We proceed to expand the problem domain of the search every $\frac{n}{k}$ generations until we are at generation n, having actually employed the original fitness function for a size k test suite only for the last $\frac{1}{n}$ generations.

We found that this approach works well in case the time of each of these evolutionary phases does run long enough to find reasonable results but not long enough to fully converge. The evolutionary search thus hits a point where it has a rough idea about the best single test sequence but still has multiple open options. At this point, it proceeds to search for a larger test suite, with limited option for the first spot of a test suite. Using the phases extension, we could cut the total amount of goal evaluations roughly in half, since the average test suite is only $\frac{k+1}{2}$ test sequences in size throughout the course of evolution.

5.2.3 Penguin Extensions

The second extension targeted two points: (1) "merging" two test sequences into one within the *combine* function and (2) applying meaningfully small changes in the search domain of test suites within the *mutate* operator. The main problem of both is the handling of test case dependencies: Two test sequences ts_1 and ts_2 cannot simply be combined by attaching the tail of ts_2 to the head of ts_1 (as in the traditional one-point crossover operation [ES03]), since the configurations in the second half of ts_2 might not conform to those of the first half of ts_1.

For solution we utilized a method we call *penguin recombination* instead. Its name is inspired by an imaginary instance of our evolutionary algorithm being used to compute the evolution of animals, where dependencies in combination and mutation can be observed as well. If we consider two test sequences as different species such as a parrot and a fish, we notice that they cannot meaningfully recombine through crossover; but, inspired by nature, we can at least evolve the parrot to another bird that is most similar to the fish, resulting in perhaps a penguin. For applying this metaphor to the test sequences considered, we utilized a notion of similarity which I introduced in Sect. 4.4.2. Note that in general every reasonable similarity notion between paths might work.

Penguin Recombination. Using this path similarity notion we suggested to combine two test sequences ts_1 and ts_2 as follows: We cut a part of the beginning of ts_1 at a random length, resulting in the incomplete test sequence ts_1^A so that $ts_1^A; ts_1^B = ts_1$ for some ts_1^B. There now exist multiple paths that may follow, of which ts_1^B is one possibility. Of all the possibilities $ts_1^B, ts_1^{B'}, \ldots$ within the current configuration at ts_1^A, we compute their similarity to ts_2^B, which is the second part of ts_2 after cutting off ts_2 at the same length as ts_1. We choose the most similar completion $ts_1^{B^*} \sim ts_2^B$ to produce a new test sequence $ts_3 = ts_1^A; ts_1^{B^*}$. This test sequence has a similar setup as ts_1 but after a certain point tries to mimic as many features of ts_2 as possible, i.e., become as much of a fish as a parrot can.

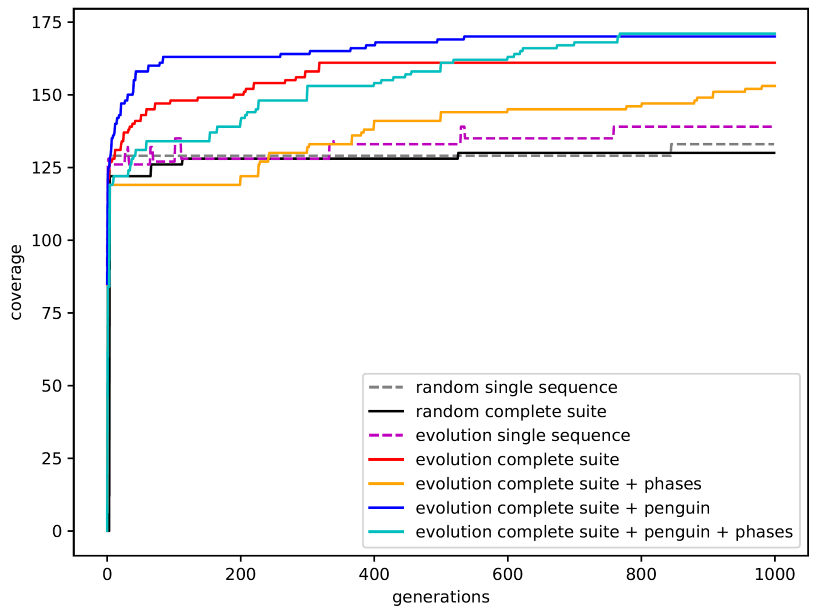
Penguin Mutation. The mutation operator had been implemented analogously, almost as a recombination of a test sequence ts with itself. We cut off ts at some random point, resulting in $ts^A; ts^B = ts$. Furthermore, we cut off the first test case of ts^B, resulting in $ts^B_{\text{orig}}; ts^C = ts^B$. We then add one test case at random to ts^A, which we name ts^B_{rand}, and make sure that $ts^B_{\text{rand}} \neq ts^B_{\text{orig}}$. From that point on, we complete $ts^A; ts^B_{\text{rand}}$ by generating the test sequence $ts^{C^*} \sim ts^C$. The mutated test sequence $ts' = ts^A; ts^B_{\text{rand}}; ts^{C^*}$ is returned with only a single test case changed and afterwards trying to mimic the original ts as closely as still possible.

We argued that this is the minimal (and still general) mutation one can implement for the domain of test sequences.

5.2.4 Evaluation

For evaluation we applied our approach to a dependency graph we recorded from simulating the self-organizing production cell. We tested a standard evolutionary algorithm evolving a test suite as well as both of our extensions individually and their combination. We also ran baseline experiments using random search as well as using an evolutionary algorithm that only evolves a single test sequence. We used a population of size $m = 50$ evolving for $n = 1000$ generations. We produced test suites of (eventual) size $k = 10$ from our test data. We chose $r_{\text{recomb}} = 0.3$ and $r_{\text{mut}} = r_{\text{hyper}} = 0.1$ for the hyperparameters providing a lot of random exploration to the algorithm favoring generality of our results over sample efficiency. The total computation time of all evolutionary processes included in the test was 1.6 hours on a machine with an Intel Core i7 processor at 2.9 GHz and 16 GB of memory.

The results are shown in Sect. 5.2.4. It can be clearly seen that the phases extension eventually achieved very similar results to both non-phase-based variants, but with considerable savings in computational resources. Furthermore, it is also evident that both penguin variants outperformed their non-penguin counterparts.



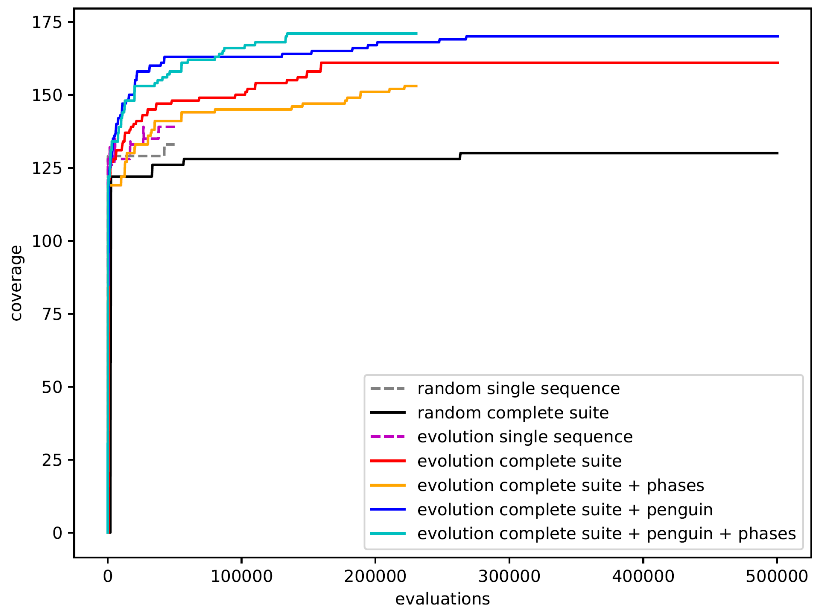


FIGURE 5.3: Performance evaluation results

Again, this validates our approach and shows that the additional knowledge given to the algorithm in form of the similarity function payed off with better end results.

5.3 Towards Assessing the Criticality of Faults

Compared with the representativity-based approach considered before the major difference of the mutation-based test goal considered in this chapter is that not only the kind of faults – anticipated and represented by mutation operators – but also the supposed severity of their effect is considered.

On the one hand it could be argued that this enables an even better prioritization of test sequences, since critical faults in terms of their effect should be eliminated with higher priority than not that critical ones. On the other hand considering not only the kind of faults but also their effect complicates test design as the only way to evaluate the effect to the environment is execution.

The short cut we were able to apply in Sect. 4.3.4 can not be applied anymore and thus, considering HOM now actually means to exhaustively generate and execute all possible mutants (which means all combinations between all considered mutation operators) with all test sequences. Since this is impractical (as shown in Sect. 4.3.3) this chapter considered merely FOMs up to here.

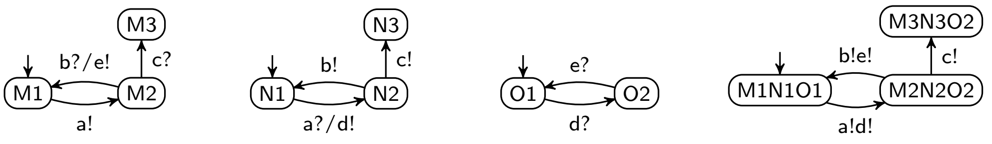
Reporting about an experiment we did for testing a "traditional", i.e., a not self-adaptive system this section considers a promising branch of future work that could help to overcome this limitation. It is building on the hypothesis that it is possible to break down the criticality of failures to faults. If it was, i.e., the hypothesis holds, we can renounce investigating the effect of each mutant with an environmental simulation but work on a model again.

5.3.1 Risk-based Interoperability Testing

The experiment considered a method for *Risk-based Interoperability Testing using Reinforcement Learning* [Rei+16]. Striving to test whether all the components of a distributed system are able to communicate with each other and thus render requested services correctly through interaction [Che13] we thereby struggled with the typically high number of possible interaction scenarios (e.g., combination of messages) which turns this kind of testing a complex task. Since it seemed impossible to cover all scenarios, we found that their relevance for being tested has to be prioritized somehow. In [Rei+16] we proposed a method to do so in a risk-based manner.

In more particular we combined *model-based* and *risk-based* testing methods with *reinforcement learning*. Thereby our approach built on given behavior models of the interacting components of the SuT, common implementation faults, and a set of the most critical failure situations, each of them described by the combination of component states and a score of its deemed effect.

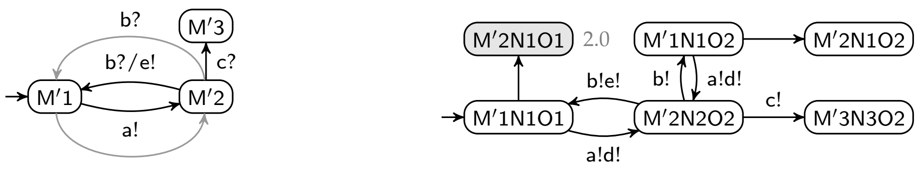
Figure 5.4 shows exemplary models for components M, N, and O as well as their composition, i.e., a *system behavior model* (please refer to [Rei+16] for the used test model specifications).



(A) Component M (B) Component N (C) Component O (D) System behavior model of M, N, and O

FIGURE 5.4: Model of the components M, N and O and their broadcasting composition.

Within those models of intended behavior no unwanted state which we called failure situation can be reached as the models do not describe faults (a combined state is assumed to be reached if all of the combined component states are active at the same point in time). By explicitly introducing common implementation faults



(A) Mutated component M′ (for M) (B) Composition of M′, N, O

FIGURE 5.5: Component test model based on the fault model of message losses and resulting system test model composing M′, N and O. Depicted in gray in (b), the system test model has been complemented by a negative situation M′2−O1 with negativity score $\nu = 2.0$.

into the models – in experiments we considered (1) message losses, (2) wrong messages, and (3) delayed messages – the situation however changes. As it can be seen in Fig. 5.5b we are indeed able to assess the *criticality* of faults, i.e., their "ability" to cause the effect of failures given an erroneous component model like Fig. 5.5a.

Given such models including anticipated faults we proposed a novel, *criticality*-based test goal aiming to maximize the *criticality* of covered faults. Since this relevance of faults is derived from the failure situation's *negativity*, i.e., a quantification of effects of failures, this test goal can be actually seen quite similar to this one discussed in the present chapter before: We strive to find a test suite that is supposed to reveal those of anticipated faults constituting maximum risk.

Note that in case of one introduced fault the extended model represents a FOM as the represented mutant is generated by a one-time application of a mutation operator. By this model-based approach we could, however, easily simulate higher order mutant also by just introducing more than one fault in the models.

5.3.2 The Malicious Developer Metaphor

Given this model-based setup the basic challenge is thus to break down the negativity of failures to the criticality of faults.

Let us, in search of a solution, imagine a malicious developer of a component actually trying to induce the whole system to reach the maximum effect of failures in real operation by implementing the "right" component faults. These faults would be the ones to be tested for with highest priority. What the malicious developer could do is to use a learning technique, such as reinforcement learning [SB98], on a simulated environment: He could implement his component as an intelligent agent which makes its own local decisions to achieve the global goal of reaching the most critical failures.

This agent then would map received rewards to the preceding actions (either specified in the behavior model or faults) so as to assess the expected return for every possible action. The ultimate reward to be reinforced would be reaching a critical failure situation. Then the agent's learned expected return for executing a fault can be understood as the fault's criticality.

For finding those test cases which cover the most critical faults with highest priority, it seems reasonable to apply the same technique as our imaginary malicious developer. This procedure can be seen as defending the system against the faults he could inject.

After the learning phase – learning which faults mean the highest criticality or reward – each agent contains a function that maps its actions to their expected return, i.e., their relevance. From a *mutation testing* perspective, the actions representing anticipated faults can be seen as mutants of the specified actions in the behavior models. The functions of the agents weight these mutants by their criticality – they break down the criticality of failures to faults. Thus, test cases can now be prioritized by the criticality of the mutants they are assumed to kill.

5.3.3 Formalization as MDP

The malicious developer's task is forming a Markov Decision Process that is quite similar to this of the tester (see Sect. 3.3.2):

We describe the malicious developer's MDP by state space S and an action space A which can be directly taken from the model composition; a map $T : S \times A \times S \rightarrow [0,1]$ giving probabilities over state transitions, such that $T(s,a,s')$ indicates the probability that action a in state s leads to state s'; and a reward function $R : S \times A \times S \rightarrow \mathbb{R}$ denoting rewards for taking particular transitions.

Since in experiments we did not assume that transition probabilities of the SuT are known by the tester, we supposed T for every state $s \in S$ and every action $a \in A$ to be uniformly distributed over the target states $s' \in S$ that are forming transitions (s,a,s') of the underlying model. In the malicious developer's MDP R reinforces transitions (s,a,s') with the negativity score of s' if s' is a negative situation, and with 0 otherwise.

In contrast to the typical task associated with an MDP, i.e., finding a policy for selecting appropriate actions in any given state w.r.t. the agent's expected return (see Sect. 3.3.2), this of the malicious developer seems rather different. Instead of finding a path through the system test model (or the MDP) that is supposed to offer the maximum reward, we first of all aim to assess the criticality of every action in order to eventually form a risk-optimized test suite. In terms of an MDP, we are searching for the expected returns of all actions.

Fortunately, some of the reinforcement learning methods originally designed for deriving optimal policies (cf. Sect. 3.1.4) do also provide us with these expected returns as kind of intermediate results. They are based on estimating action-value functions, i.e., mappings of states (or state-action tuples) to the expected return when being in the given state (or selecting the given action in a given state) [SB98]. Thus, in using one of these algorithms, we are able to estimate the actions' criticality by sample experience.

5.3.4 Q-Learning

In *Q-Learning* we decided to go with one of the most prominent representative of those action-value-based RL methods in our experiments. As kind of *temporal difference learning* it offers the special charm of working on sample experience and thus not requiring a model. In this method, we thus do not have to explicitly build the system test model which may be prohibitively large due to the number of components and possible faults.

In order to exploit this advantage, we chose in our experiment a fully decentralized approach: Within a simulation of system runs, we suggested to associate each single component of the SuT with an agent that learned the expected return of its actions.

Each agent owns a so-called Q-function mapping environmental states together with actions to their expected return. We called the pair of an environmental state and an action a *decision*. After a reward R_{t+1} is received for action a_t executed at time step t out of the environmental (global) state $\bar{s}_t$, the expected return for the decision $(\bar{s}_t, a_t)$ is getting updated as follows:

$$Q_{t+1}(\bar{s}_t, a_t) = Q_t(\bar{s}_t, a_t) + \alpha\big(R_{t+1} + \gamma \max_a Q_t(\bar{s}_{t+1}, a) - Q_t(\bar{s}_t, a_t)\big) \ . \tag{Q}$$

The parameters $\alpha \in \,]0,1]$ and $\gamma \in [0,1]$ denote the *learning rate* and the *discount factor*.

Decisions which are not mapped on an expected return yet get a default assignment of 0. As one can see in (Q), the expected returns – that are representing our measure of criticality – are updated with respect to a policy in which the agent chooses anytime the action with the highest criticality (represented by the max-term in (Q)). This *optimal policy* invokes the worst-case behavior of the component, that, as we supposed, is the most appropriate one in case of risk-based testing.

5.3.5 Deriving Test Cases From Q-Values

After the learning phase we have a set of agents, each one containing a Q-function mapping decisions to criticality values. Recall that this, as an intermediate result, would enable the imaginary malicious developer I mentioned before to implement the most critical faults in his component; and it enables us to assess observed decisions of any of the SuT's components in real operation. However, we still want to use this learned information for generating test cases covering the most critical faults. For this purpose, two things have to be considered:

(1) Positive test cases, as we exclusively considered in our experiments, do only include decisions with specified actions (*specified decisions*) but are able to detect implemented decisions with mutated actions (*mutated decisions*). More precisely, we assume the test of a specified decision to detect all of its mutants, i.e., decisions with the same state but with actions that are mutants of that contained in the specified decision. Hence, we have to distinguish between a decision's criticality and a specified decision's relevance for being tested that, in fact, should even comprise the criticality values of its mutants.

(2) A Q-function, as we formed it, assesses decisions with local actions (*local decisions*). A system test case, however, should specify the execution of *global decisions* involving one local decision per component.

Thus, we aggregate the learned criticality values by (1) local and (2) global relevance functions whereby the latter depends on the first. System test cases are generated and assessed using the global relevance functions.

Local Relevance Function We aggregate for each specified decision d the criticality scores of mutated decisions $M(d)$. As result the local relevance function r of an agent maps each specified local decision to its relevance for being tested. In concrete terms we define the relevance of a decision by the sum of the criticality values of its mutants. This seems reasonable for testing, since a specified decision is deemed to reveal all of its mutants if they are implemented.

From a mutation-based testing perspective, the relevance can be seen as the reward for killing a set of mutants. More formally, for a specified decision $d = (\bar{s}, a)$, let $M(d)$ be the set of mutated decisions whose actions are mutants of a and whose composed state is $\bar{s}$. Then we proposed to define the local relevance function by

$$r(d) = Q(d) + \textstyle\sum_{d' \in M(d)} Q(d') \, .$$

Global Relevance Function By the global relevance function $\bar{r}$ we map each *possible* global decision to its relevance for being tested. A global decision $\bar{d}$ consists of one specified (local) decision per agent. In experiments we said that a global decision is possible iff the contained local decisions can be made at the same time. Thus, the local decisions contained in a possible global decision share the same composed state and actions which satisfy the chosen communication paradigm.

Since only the specified local decisions are considered, but not their mutants that would result in much more possible global decisions, the computation of the set of global decisions turned out to be feasible, even for rather complex models. Note that the execution of a global decision by a test case implies the execution of all included specified (local) decisions. We defined a global decision $\bar{d}$ to be as relevant as the sum of its local decisions:

$$\bar{r}(\bar{d}) = \textstyle\sum_{d \in \bar{d}} r(d) \, .$$

Building on this global relevance function, we are able to derive a *risk-optimized test suite*, i.e., a suite of a desired number of logical interoperability test cases that covers as much relevance as possible then by a simple algorithm (please refer to [Rei+16] for more details).

5.3.6 Towards Application for Testing Self-Organizing Systems

Having discussed these experiments let us now consider how this method could be applied to our setting with the self-organizing production cell described before.

Indeed we do not have the behavioral models for components which we had in reported experiments, but instead we have a simulation letting us estimate the effect of faults that are introduced by mutation operators. What we can do thus is activating faults by mutation operators in simulation and logging the effect – similar as we did before by the so-called label vectors. Failures are backpropagated through the sequence of faults as previously discussed for actions in the Q-learning procedure. The Q-table assessing actions, i.e., activating particular mutation operators $o \in O$ in particular system states observed in simulation, that represent the mutation operator's fault criticality thereby evolves step by step.

Instead of evaluating each test case of each test suite with each mutant we would thus construct the Q-table by a sequence of random tests with random mutants. The Q-learning's dynamic programming intention thereby grants us convergence towards the optimal values. The relevance, i.e., the worthiness – or as we called it before the *relevance* – of particular test steps $ts(t)$ with $t \in [1..|ts|]$ then results from the criticality of covered mutation operators[1].

Still, due to the Q-learning's max operator the computed value for a particular fault has to be rather considered as worst case regarding all the possible HOMs it could be part of than an average. As discussed before this might be however exactly the case which we want to exclude with highest priority. Nevertheless also experiments with other aggregations seem promising as the essential question for future work will be how to efficiently construct adequate test suites of sequences given the learned Q-table.

Still, we have to note that since the algorithm proposed in [Rei+16] was built on behavioral models of the SuT's components it will not be directly applicable for self-adaptive systems for which we do not have such models given. Again, the use of meta-heuristical search approaches such as evolutionary algorithms appears a gainful try.

5.4 Related Work

In reported experiments we considered test strategies for a novel mutation- and risk-based test goal. While also in related work it seems quite common to use fault-based techniques for evaluating the quality of test suites [BMK04; Rot+02; Zha+11], the huge majority of approaches, including the cited ones, applies other test goals for actual generation. This might be due to the high costs for goal evaluation, which we were able to reduce by the *phases* extension.

Here, we made use of search-based testing techniques [McM04] for test design w.r.t. the introduced goal. In particular, we made use of an evolutionary approach for finding adequate test suites. Indeed, this kind of algorithms is yet actively used in the automatic generation of test cases for software [McM04; WL05]. In recent years, the research community considered the issue of *whole test suite generation*, in which the aim of applying an evolutionary algorithm is not to find the most important test cases but instead to find the ideal combination of test cases that make up a concise but approximately complete test suite for a given software [FA11].

Especially for this whole test suite domain we proposed the *phases* as well as the *penguin* extension which were shown to improve our test design. Some general approaches on evolutionary algorithms have also already introduced dynamics into originally non-dynamic problems in order to improve the quality of the search result [Urs02; GBLP18]. These also use measurements related to the similarity between individuals in their evaluation, which may then change over time as the population changes. Similarity has been incorporated into the recombination process e.g. in [IS03], though on a different level than in our approach, viz. at the level of mate

[1]Whether or not a mutation operator is covered might be assessed by constructing a FOM m from program p and comparing the effect of $p.execute(ts(t))$ and $m.execute(ts(t))$. Note that still we do not have to construct every HOM in this case.

selection mirroring biological evolution. The advantages of such approaches for test design had been, however, not considered yet. Also, the challenges for testing self-organizing systems were not addressed by evolutionary algorithms before.

5.5 Conclusion

Apart from the novel mutation-based test goal and the evolutionary approach on test design we sketched a promising path for future work that might enable considering also higher-order mutants in future. A first experiment considering a non-adaptive system under test had been presented in Sect. 5.3. Adapting this setting to testing self-adaptive systems will constitute an important part of future work.

Further, as the initial evaluation of the presented algorithm just considered a single concrete case, testing a self-organizing production cell, we expect to be able to generalize our findings in future. Applications to be considered include code-level test sequence and test suite generation. I envision several cut points with practice-oriented applications, such as test suite minimization and construction for distributed systems, waiting for being explored.

Chapter 6

Continuous Test Models

Up to here we have considered classical test strategies for reactive systems in Kapitel 2 and elaborated the particular challenges when it comes to testing self-adaptive systems in Kapitel 3. Kapitel 4 und 5 presented some experiments on deriving test strategies for the particular kind of self-organizing systems.

As we saw that model-based testing, and in particular FSM-based test strategies, helps improving efficiency in test design in Sect. 3.3 we extended this technique for application on testing self-adaptive systems: first, we derived a behavioral model by using a simulation of the SuT in its environment; and secondly, we based the search for a test strategy on this. The presented strategies of test policy learning and test policy evolution instantiated this technique considering hierarchical FSMs and dependency trees as underlying models.

The findings suggest that given one of those models, let it be anticipated or learned, we will be able to find an adequate test strategy. But what if we cannot derive such a model? What if the SuT's state space does not allow us constructing an FSM? Those are exactly the questions the following sections are meant to address.

Present chapter considers in more particular capturing self-adaptive system behavior by continuous test models and forming test strategies for them. By the term continuous I thereby refer to the state space which here might include continuous variables not letting us categorically separate states.

While in classical conformance testing this separation is usually driven by forming equivalence classes on inputs and states I here assume that due to the sheer number of variables this is not always convenient for self-adaptive systems. We have to assume that we do not really know a priori the peculiarities of the environment determining the system's response, which implies that we should initially consider all of them – in Sect. 3.3 I proposed to do so by using an environmental simulation.

Section 6.1 reports about experiments we did for capturing adaptive agent behavior by *Artificial Neural Networks* applying a well-known technique for handling such continuous state spaces. Based on an adequacy criterion for this kind of test models (Sect. 6.2) a test strategy is presented that builds on the well-known machine learning technique of *autoencoding* (Sect. 6.3).

The reader should note that some parts of this chapter are direct citations from papers [Rei+17] and [RK17a] which are set in context of present work.

6.1 Capturing System Behavior By Neural Networks

The experiment the following section is going to report about is meant to show that it is possible indeed to capture the behavior of adaptive software agents in artificial neural networks (Sect. 6.1.2). While it was originally motivated by the need for a function imitating all participants within a distributed, self-organizing system within a *statistical software testing* setup – we confronted an overlying reconfiguration mechanism with preferably realistic situations (Sect. 6.1.1)–, we will use the resulting insight that neural networks might be useful test models as well for further experiments later. The considered case study, a smart energy grid comprising several adaptive power plants, is presented in Sect. 6.1.3.

6.1.1 Statistical Software Testing for SO Systems

Statistical software testing applies stochastically generated sequences of test inputs to a SuT which represent its anticipated use. In doing so, this approach allows the tester to reason about the system's expected field quality [WT94] while limiting the number of test cases to execute. Further, the obtained models of the environment, in which the SuT is expected to be employed, can be used to prioritize and thus to reduce test efforts in general. As previously mentioned this reduction is of particular interest when dealing with very huge and ramified state spaces of the SuT's environment, since, without prioritization, the so called *state space explosion* is likely to give rise to test suites of uncontrollable size.

Eberhardinger et al. explained this phenomenon and consequential challenges with particular regard to a class of self-adaptive systems we yet considered – the *self-organizing systems* [Ebe+17b]. Recall that this kind of self-adaptive software systems is usually composed of an *agent layer* comprising a number of adaptive agents that are associated with their environment, and an *organization layer* including a self-organization mechanism (SO mechanism) which is responsible to reorganize the agents if necessary.

In providing an approach for isolated testing of the self-organization mechanism within the organization layer, Eberhardinger et al. instantiated the idea of statistical testing by structuring the huge environmental state space using so called *environmental profiles*. However, by means of a case study that considered a self-organizing energy grid, they found that the environment itself and the underlying probabilistic model not directly determines the searched test input sequences for the SO mechanism under test, as this exclusively depends on the agent layer.

The implemented agent behavior maps the environmental profile to an agent-specific state. Thus, they stated that in order to obtain statistically relevant test inputs, one needs to take an indirection: as visualized in Fig. 6.1 the proposed approach pipes the output of environmental profiles in stochastic test mock-ups imitating the estimated agent behavior – the so called *influence functions* – the output of which can then be applied to the SO mechanism as test input.

Though we usually have access to lots of statistical data concerning the SuT's environment, e.g., weather data or other physical models, which can be used to construct environmental profiles, it is rather unclear how to obtain the mentioned

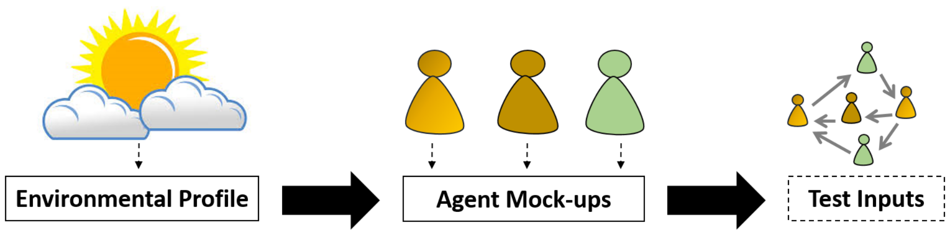


FIGURE 6.1: Proposed process for statistical testing SO mechanisms. Mock-ups simulate agent behavior in response to environmental conditions which are provided by environmental profiles. Outputs of the mock-ups serve as test inputs for the SO mechanism.

influence functions for constructing agent mock-ups from a limited set of observations. In [Rei+17] we characterized this task as follows: Given a set of logs that report the agent behavior under specific environmental conditions, we are seeking a generalization that allows us to predict the agent behavior even for previously unseen conditions.

Before regarding an implementation of such a generalization which can be seen as continuous test model as well, let us with a self-organizing smart energy grid consider the case study from [Rei+17] as a basis for evaluation.

A Self-organizing Smart Grid The wide-spread installation of weather-dependent power plants (PP) using wind or solar energy as well as the advent of new consumer types like electric vehicles put a lot of strain on power grids. To address the ris-

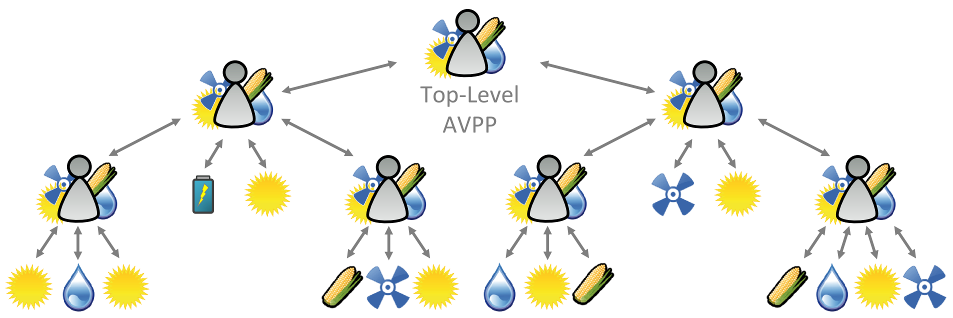


FIGURE 6.2: Hierarchical system structure of a future autonomous and decentralized power management system: Power plants are structured into systems of systems represented by AVPPs that act as intermediaries to decrease the complexity of control and scheduling.

ing challenges of future power management systems, Steghöfer et al. presented the concept of *Autonomous Virtual Power Plants* (AVPPs) in [Ste+13]. AVPPs represent self-organizing groups of two or more power plants of various types (cf. Fig. 6.2). The organizational structure represents a *partitioning*, i.e., every PP is a member of exactly one AVPP, which is established and maintained by a (partitioning-based) self-organization mechanism.

Each AVPP has to fulfill a fraction of the overall power demand in the energy grid. For this purpose, each AVPP autonomously calculates schedules for directly subordinate dispatchable PPs. The calculation of the schedule depends on different influence factors that might effect the members of the AVPP. Foremost, external influence factors, e.g., the wind condition for a wind turbine, are challenging to handle since they introduce an uncertainty into the system.

To cope with the accruing uncertainties, AVPPs autonomously adapt their structure to changing internal or environmental conditions. Thus, they are able to live up to the responsibility of maintaining an organizational structure enabling the system to operate reliably which is measured by a numeric value called *trust*. In particular, if an AVPP repeatedly cannot satisfy its assigned fraction of the overall demand or compensate for its local uncertainties, it triggers a reorganization of the partitioning.

Deriving Influence Functions The agent layer of the present self-organizing, adaptive system is formed by the various power plants and AVPPs. Affected by their environment each of these agents produces output. This output not only comprises plain energy values, but also a trust value between 0 and 1 assessing the reliability of the power plant's forecast. And exactly those trust values form test inputs for the SO mechanism, as reorganization in the proposed system is exclusively based on the notion of trust. Figure 6.3 visualizes this data flow at the example of weather dependent power plants.

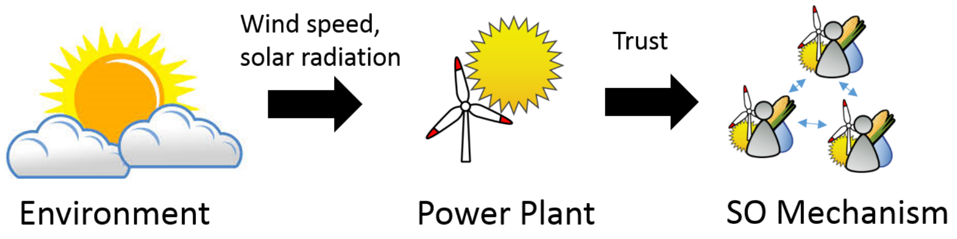


FIGURE 6.3: The output of a weather-dependent power plant is influenced by environmental conditions, such as wind speed and solar radiation. A trust value signals the estimated reliability of forecasts.

The task of generating statistical tests for the present SO mechanism can thus be formulated as finding trust vectors (one trust value per considered power plant) which appear to be most likely within the real world. In order to obtain such trust vectors, we needed to consider both statistics concerning the environment as well as the assumed agent behavior.

Let us consider this process for a solar power plant. To obtain its most likely outputs we would first register weather statistics in the region our PP will be located in. These statistics could be captured within an environmental profile. Given such a profile we still need to find a mapping from environmental states to trust values the agent is supposed to return. This mapping, the influence function, can be seen as a mock-up of the real system replicating its behavior. As we have no unequivocal specification mapping weather states to prognosticated trust values for our PP, we could employ some handmade heuristics which operationalize our assumptions. For instance, we could assume that the trust in the forecast of a solar power plant descends with increasing variance in solar radiation.

Striving to reduce test effort, which arises when reasoning about heuristics, while maximizing statistical accuracy, which might suffer from wrong assumptions or subjective assessments of the tester, in [Rei+17] we investigated a more automatable

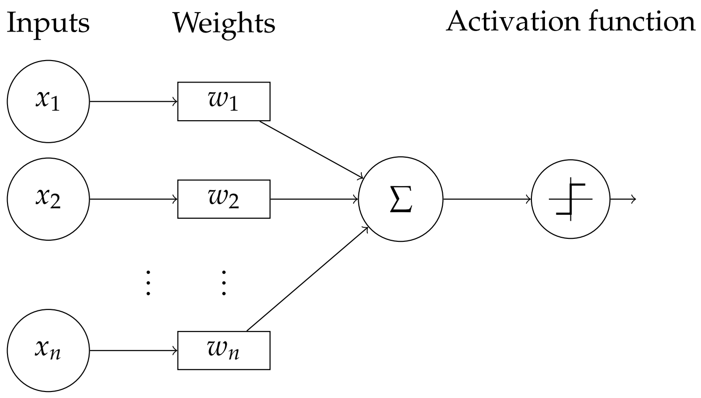


FIGURE 6.4: Inner Parts of a Neuron

approach using machine learning techniques for deriving the influence function. Inspired by the state of the art in related research fields, such as predictive maintenance or anomaly detection, we decided to solve this regression task by using and training artificial neural networks (ANNs).

We argued that this approach offers the following advantages: (1) The popularity of training such models resulted in lots of literature, frameworks and tutorials. This kind of accessible support makes the considered techniques applicable also for test engineers which are no ML professionals. (2) It was shown that ANNs are able to approximate arbitrary functions (in dependence on the meta-parameters). It is thus allowed to assume that using this kind of models we are able to estimate almost all influence functions.

6.1.2 Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANN) are computer models meant to store and to learn from presented information. Inspired by biological neurons ANNs can be used for regression and classification tasks, such as machine learning, pattern recognition, or trend analyses. While this section is meant to give a rough overview about the general building blocks and learning procedures of ANNs, the interested reader is referred to [Hay09] for a more comprehensive consideration of this kind of models.

Elements of Neural Networks All the diverse manifestations and architectures of ANNs yet considered in neural informatics or artificial intelligence have in common a set of basic elements. Each ANN consists of a set of nodes, called artificial *neurons*, and a set of *connections* between the neurons resembling human synapses.

As the schematic representation in Fig. 6.4 depicts, the input signals of a neuron are processed by first multiplying each input signal with a weight, summing all the weighted input signals up, and then applying a non-linear function called *activation function* on the weighted sum of inputs. The result is reported through the outgoing edges as inputs to other neurons or as overall output of the ANN to the caller. Learning of an ANN is performed by adjusting the weights between the neurons and edges over time.

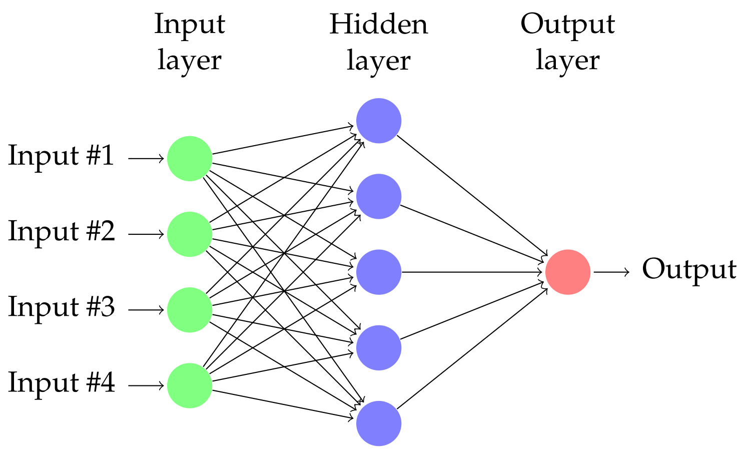


FIGURE 6.5: A Feedforward Artificial Neural Network

Feedforward Neural Networks Neurons are typically grouped by layers which are characterized by the kind of transformations of inputs, i.e., the activation function. The idea behind is that signals are traveling through those layers – from the first, the input layer, to the last, the output layer possibly running in loops.

Here, I want to concentrate on one of the simplest types of artificial neural networks – the *Feedforward Neural Networks*. In this kind of networks the information moves only in one direction, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network; see Fig. 6.5 for an instance.

Gradient Descent The so-called *back-propagation* is the most popular learning technique for feedforward neural networks. Here, some predefined error function is used to compare the network's output values with the correct answers. The output value is then fed back through the layers of the network and the weights of each connection are adjusted to reduce the error. This process is getting repeated over a number of training cycles until the error in calculations of the network is sufficiently small.

The adjustment of weights is usually done by a non-linear optimization method called *gradient descent*. Under the assumption that the activation functions used are differentiable it calculates the derivative of the error function with respect to the network weights and changes the weights in a way that the error decreases (thus going downhill on the surface of the error function).

Training in Batches Besides updating the ANNs weights after each output observed it is also common to consider the aggregated error of whole *batches* of samples for single updates. The batch size thereby defines the number of samples to be presented to the ANN before adjusting the weights. If the batch size equals the size of the training set, we are talking about the special case of *batch gradient descent*. While *stochastic gradient descent* considers the inverse extreme with a batch size of 1, the so-called *mini-batch gradient descent* means a batch size in between.

6.1.3 Imitating a Smart Grid with an ANN

Though in experiments we had access to a fully integrated simulation environment of the aforementioned self-organizing system, for evaluation purposes, we pretended to have only given the following artifacts: *Environmental profiles* which model the most likely weather conditions for a considered area within Markov chains, as well as *log* sequences for a set of particular solar power plants describing their output at a given time in response to particular environmental states.

Note that the evaluation of experiments was exclusively concerned with estimating the effort and measuring the accuracy of the learned, predictive mock-ups emulating the behavior of considered power plants. The desired generation of statistical tests for the SO mechanism under the use of our models would additionally demand access to the SuT itself as well as some kind of test oracle. As our methodology can be easily plugged-in into the process described in [Ebe+17b], the interested reader is referred to this work for more details on those mentioned test artifacts.

Heuristical Influence Functions as Baseline In experiments we based the search for adequate heuristics on the following argumentation: In general, an influence function is meant to map one environmental state to the expected output of the considered agent. In the present case, this would mean to map the weather conditions (we exclusively considered the prevalent wind speed and solar radiations) at a particular time step t to the trust value of a weather dependent power plant observable at time step $t + 1$. However, even if it seems natural that the performance of a weather dependent power plant depends on the weather one time step before, such a dependence seems not adequate for the trust value, as this only assesses the quality of a forecast.

This forecast refers to the difference between the expected and the actual output. It seemed likely to us that in case of the weather dependent PPs, the energy output forecast depends on the quality of accessible weather forecasts. As these were, however, unknown to us, we introduced another assumption: the quality of weather forecasts might depend on the weather conditions observed at the last n time steps: the more variance in weather, the more unreliable the forecast. Another assumption was, that the loss of the weather forecast's accuracy can be approximated by a constant, whose value however might differ between the different power plants and locations.

Overall, we considered the following heuristics:

Heuristic 1 Constant trust value at 0.8. This constant had been chosen based on the assumption that weather forecasts might be inaccurate in 20% of cases.

Heuristic 2 Trust value at time step $t + 1$ depends on weather conditions at time step t. The worse the weather conditions for the energy output of the PP, the worse the estimated trust.

Heuristic 3 Trust value depends on variance within the weather conditions of the last 5 time steps. The more variance, the worse the estimated trust value.

Learning an Influence Function Apart from the mentioned heuristics, we trained an ANN to represent the influence function. By informal search we decided for a feedforward neural network comprising three hidden layers with 256, 128, and 64 neurons. The training set, i.e., the sample data provided to the network was gathered as followed: We generated logs by observing the agent behavior in response to randomly generated environmental profiles within our simulation. Each sample thereby included the last 5 states of environmental profiles as input as well as the reported trust value as output. These logs then were used for training. For this, we utilized a kind of on-the-fly batch training by successively increasing the training set with alongside generated logs, instead of writing the whole bunch of logs in a file. Each iteration over the growing training set was thereby counted as one training *epoch*.

With the so-called *Mean Absolute Error (MAE)* a common error function defined by

$$MAE = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

was used for measuring accurancy; n is the number of predicted values, y_j is a value predicted by the ANN, and $\hat{y}_j$ is the actual value. The neurons of all hidden layers were equipped with the *rectified linear* activation function, i.e., $f(x) = \max(x, 0)$.

Figure 6.6 shows the resulting learning curve we observed. One can see that training progress is rather low in the first phase (450 epochs), but suddenly grows in the following. After 500 epochs of training we achieved a mean absolute error of < 0.1 on randomly generated environmental profiles.

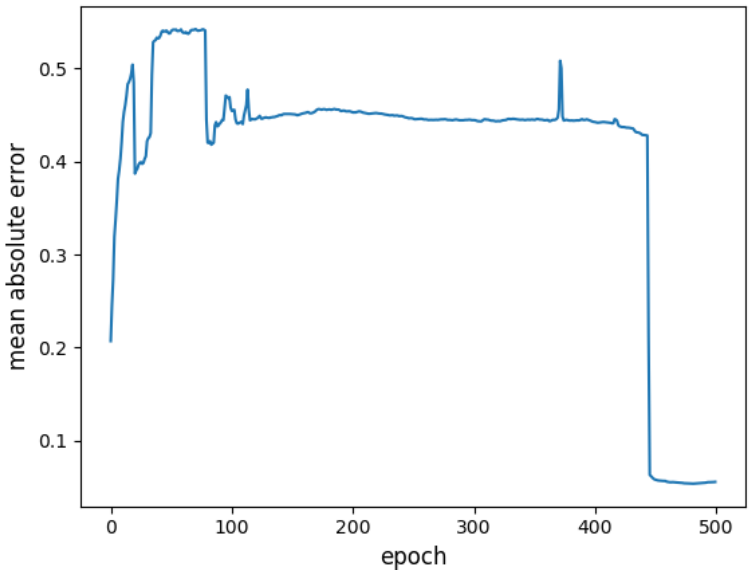


FIGURE 6.6: Learning curve describing the trend of mean absolute error with increasing training time.

To compare these results to the presented heuristics, we evaluated all of them together with the trained ANN on a separate run of 100 time steps simulating the environmental profiles Eberhardinger et al. used in their previous work [Ebe+17b] for evaluation. Table 6.1 shows the results. Note that we did no parameter fitting on the heuristics as we assumed that the accuracy of a handmade heuristic can be

KPI / Methodology	Learned Model	Heuristic 1	Heuristic 2	Heuristic 3
mean absolute error	0.08	0.3	0.3	0.25
standard deviation	0.07	0.08	0.17	0.18

TABLE 6.1: Comparison of ANN results with different heuristics.

arbitrary optimized at the expense of time. Our intention – to show that this job can be outsourced to an ML algorithm – was confirmed by the high accuracy of the ANN.

6.2 Compressing Uniform Test Suites

As the latest section suggests it is possible to capture (part of) the behavior of adaptive software systems by artificial neural networks. However, the critical task is still finding test strategies based on those insights. According to the test process depicted in Fig. 3.5, how can an ANN be used in order to derive test inputs for testing a SuT in reality? What we certainly need first is an adequacy criterion marking the kind of test suite we want to have. With the *Compression Coverage Goal* this section introduces one we found in [RK17a] resembling typical coverage criteria from models like FSMs for trained neural models.

The basic hypothesis was that the specification of self-adaptive system behavior is formalized by invariants over the system and environmental state such as the corridor of correct behavior we already considered in Sect. 4.1.3. For this we supposed the SVS described in Sect. 2.1.2 a good instance. Its behavior should be characterized by the goals that the robot should drive directly to a user-given mark, without falling down the stairs which lets us directly derive the two invariants for testing: (1) The robot never falls down the stairs and (2) it reaches a user-given mark as quickly as possible.

As this instance shows, we rather aim to exclude unwanted SuT behavior than to investigate the conformance of the observed behavior with specified solution approaches as by classical FSM-based testing. Each test case strives for the same goal, i.e., breaking the invariant. As resulting test suites then consist of test cases that only differ in test inputs but not in test goals in our experiments we named them *uniform test suites*.

6.2.1 Uniform Test Suites

Since neural (test) models are assumed to be trained with exemplary SuT executions in simulation, deriving test cases from this kind of test models can be seen as kind of test suite reduction, were the set of samples with which the model was trained are the original – and the set of test cases later derived are the reduced test suite.

We found that in contrast to classical test suites comprising test cases with diverse execution and success conditions, uniform test suites can be reduced by just choosing a subset of representative test inputs. And we noticed that uniform test suites could offer a special chance: Instead of selecting a subset of test cases by test suite reduction, uniform test suites can be used to generate inputs which are not necessarily included within the original suite while still retaining the original intention. In comparison to traditional test suite reduction this kind of *test suite compression*, as we named this particular reduction task, thus offers the special chance that it provides the tester with completely new insights.

Let us consider an observable an executable simulation of the SVS. The simulation exposes a state space S. While the simulation is running, its momentary state $s \in S$ is modified by actions $a \in A$ that are executed by the SuT. Furthermore, the tester is able to establish a particular state through test inputs $tc \in S$ just before a run as source state.[1] A uniform test suite $TS = (TI, inv)$ comprises test inputs $TI \subseteq S$ as well as an invariant $inv : S \to \{tt, ff\}$ which signals hazardous states by mapping them to ff.

The test inputs are used to gain confidence that inv holds (i.e., $inv(s) = tt$ for all reachable states $s \in S$). Each test input triggers particular behavior of the SuT which can be observed in the form of sequences of system actions (action sequences) in simulation. For the sake of simplicity, we assumed action sequences to be ordered in discrete time steps $t = 0, 1, \ldots, n$, in each of which the simulated system selects an action a_t based on the current state $s_t \in S$ using a deterministic policy $\pi : S \to A$. The resulting simulation state is supposed to be determined by a transition function $T : S \times A \to S$ when action $a \in A$ is executed in state $s \in S$.

Based on those assumptions and a discount factor $\gamma \in [0, 1[$, we define the *behavioral distance* Δ^π of test inputs tc_1 and tc_2 as follows:

$$\begin{aligned} &\Delta^\pi(tc_1, tc_2) = \textstyle\sum_{0 \leq n} (1-\gamma)\gamma^n \delta(\pi(s_1^n), \pi(s_2^n)) \\ &= (1-\gamma)\delta(\pi(s_1^0), \pi(s_2^0)) + \gamma \Delta^\pi(s_1^1, s_2^1)\ , \end{aligned} \tag{6.1}$$

where $s_i^0 = tc_i$ and $s_i^{k+1} = T(s_i^k, \pi(s_i^k))$. This behavioral distance builds on an action distance function $\delta : A \times A \to \mathbb{R}_{\geq 0}$ which is meant to quantify the differences of actions $a_1, a_2 \in A \times A$. Though there might be more sophisticated implementations, let us resolve this for the time being by $\delta(a_1, a_2) = 0$ if $a_1 = a_2$ and $\delta(a_1, a_2) = 1$ otherwise.

6.2.2 The Compression Coverage Goal

Compressing a uniform test suite $TS = (TI, inv)$ to $TS' = (TI', inv)$ with a maximum number of n inputs aims at substantially reducing the number of test inputs to apply ($n = |TI'| \ll |TI|$). As we in our experiments intended to maximize confidence that the invariant inv holds for every conceivable input, we furthermore aimed at producing as diverse behavior of the SuT as possible. The aims of minimizing the number of test inputs and maximizing the manifoldness of produced system actions are, however, conflicting, since the number of actions the SuT is triggered to execute decreases with the number of applied test inputs.

Solving this conflict as well as possible for a given maximum of n inputs requires to find a set of test inputs TI' with $|TI'| \leq n$ that covers as much of the SuT's behavior

[1]The reported experiment assumed that the tester is able to directly modify the state space in simulation. Thus, there was no separate actions space introduced for the tester, but a test input considered to be a particular state.

that was triggered by the original inputs TI as possible. Referring to the defined behavioral distance function Δ^{π} on test inputs, we conceived this kind of *compression coverage goal* as the goal of minimizing the following function with an adequate TI' with $|TI'| \leq n$:

$$\Delta^{\pi}_{TS}(TI, TI') = \max_{tc \in TI} \min_{tc' \in TI'} \Delta^{\pi}(tc, tc') \ . \tag{6.2}$$

This function can be interpreted as the distance of two sets of test inputs, determined by this test input of the original suite whose triggered behavior of the SuT is covered worst by the new, compressed test inputs. The lower Δ^{π}_{TS} for a compression, the broader the maintained behavioral coverage and thus the higher the fulfillment of the compression coverage goal.

If all test inputs of the original suite vastly differ in terms of the produced SuT behavior, (6.2) will be hard to minimize. It can be assumed that choosing n random inputs then leads to similar results like choosing them based on structural investigations. We however claim that test suites naturally include redundancy, as the testers who define the inputs follow certain assumptions that result in deeply coded patterns of test inputs. Considering for instance the SVS testers could assume that the robot drives straightly to placed dust particles. If this is the case, they would probably place dust particles next to the stairwell for testing if the robot falls when driving to the dust. This would cause a pattern of similar test inputs within a test suite testing for the invariant that the robot does not fall.

6.3 Test Suite Compression with Variational Autoencoders

If test inputs of the original test suite, indeed, include deeply coded patterns, it would be helpful to recognize those for compression. Distributing the n inputs to choose over the patterns would promise higher compression coverage than random choice which could in the worst case lead to n inputs of the same pattern. Since seeking such deeply coded patterns in high-dimensional data — as our test inputs comprising contextual features are supposed to be — is a well-known task in other domains such as image recognition, it appears fruitful to reuse approaches found there for our task.

In image recognition, it is common to train neural networks as deep models for the input media. These networks are able to generalize over concrete features of the inputs and thus to classify them. This classification of inputs can either serve directly as output, or it is used as intermediate result for generating data that is similar (in terms of the same latent class) to the presented inputs. Models serving for the latter case are called *generative models* [NJ01].

For test suite compression, generative models are of particular interest as we want both to extract the deeply coded patterns of the original test suites' inputs and to generate compressed suites comprising n representative inputs. *Variational autoencoders* [KW14], which we used for our experiments in [RK17a], offer a way for doing so.

6.3.1 Variational Autoencoders (VAE)

Autoencoders in general are neural networks constructed of two parts: An *encoder network* which transfers the high dimensional input data to a compressed representation; and a *decoder network* which takes such compressed representations of data and therefrom reconstructs the original inputs. Figure 6.7 depicts this scheme. The illus-

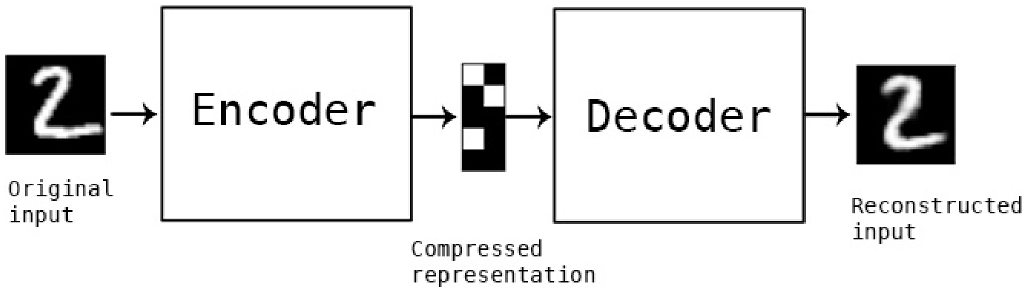


FIGURE 6.7: Autoencoder scheme.

tration is taken from [Cho16] which also offers a convenient tutorial for coding various types of autoencoders (including variational autoencoders) with Keras [Cho15].

Trained to model the identity function for exemplary inputs, autoencoders are normally used for data denoising [Vin+08; Ben+14] as well as for dimensionality reduction [HS06]. Nevertheless, classical autoencoders are rather not generative. As there is no uniform semantics for the latent parameters, the quality of inputs generated for previously unseen compressed values is contentious. *Variational autoencoders* fix this by adding semantics to the latent space: they are learning the parameters of a probability distribution modeling the training data. Figure 6.8 depicts the particular structure of variational autoencoders.

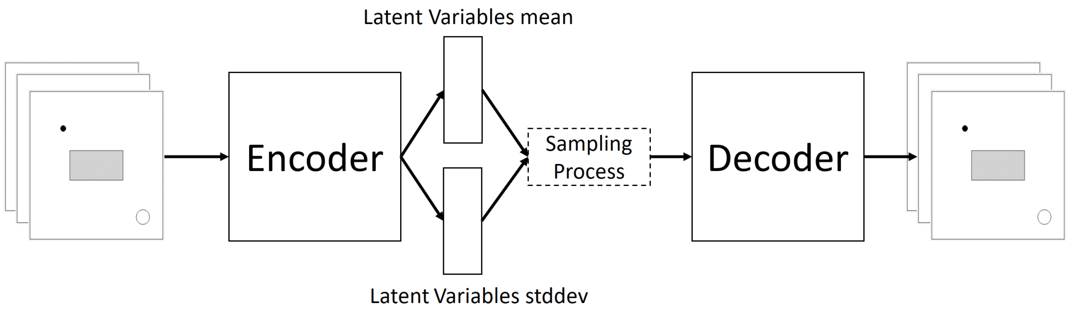


FIGURE 6.8: Variational autoencoder scheme.

Each of the training samples is mapped to the latent space comprising the two variables *mean* and *stddev* (the dimensionality of *mean* and *stddev* can be arbitrarily chosen). These variables are the basis for a latent normal distribution from which similar data is generated by, e.g.,

$$z = mean + stddev \cdot eps \,,$$

where *eps* is a random normal tensor of the same dimensionality as *mean* and *stddev*. The generated data z is then passed to the decoder network that is supposed to map it back to the original input. VAEs are usually trained by the use of error functions that are composed of a generative reconstruction loss between the original and the reconstructed input (e.g., mean-squared-error or binary cross-entropy) and the Kullback-Leibler divergence of the latent variables with the Gaussian distribution. More detailed explanations including the mathematical basis of VAEs can be found in [Doe16].

So far, VAEs in different variants have been successfully used for generating images, such as handwritten digits or faces [KW14; SKW+15], as well as for predicting trajectories of pixels in a scene [Wal+16]. There had also been experiments with generating objects in scenes [Kul+15]. Using VAE for capturing and generating test inputs contained in (uniform) test suites is a so far unnoted area of application.

6.3.2 Training a VAE with Test Inputs

The idea of experiments was to train a VAE with all the test inputs that are included in a given uniform test suite, in order to obtain a generation process producing inputs for arbitrarily sized compressions. This seems reasonable, since test input generation resembles previous application scenarios of VAEs, such as generating images that are similar to presented ones.

In [RK17a] we evaluated this idea with the help of the SVS example. Utilizing a simulation of the SVS within its environment, we considered the following setting: The SVS was located within a room of $20\,\mathrm{m} \times 20\,\mathrm{m}$ which contained an arbitrarily sized rectangular stairwell. A mark could be placed anywhere within the room except the area covered by the stairwell. In order to vacuum-clean the dust, the simulated SVS was supposed to directly drive to the position of the previously placed mark — without falling down the stairs. The input space for testing this invariant included dimensions for the coordinates of the robot position, the position, width and height of the obstacle as well as the coordinates of a mark.

As described in Sect. 6.2, we assumed the test inputs included in a uniform test suite to follow certain deeply coded patterns. We used the following process to replicate test suites comprising 2000 test cases of this kind for our experiment: We constructed a set of 5 random, but with respect to the intention of a room setting reasonable test inputs. These were then used to derive 400 similar but different input vectors respectively by slightly modifying the values of dust and stairwell positions.

We implemented a VAE as fully-connected feedforward network comprising two hidden layers of 32 neurons respectively (one for encoding, one for decoding) and a two-dimensional latent space. This model was trained following the approach mentioned in the previous section.

Figure 6.9 visualizes the two-dimensional representations of our test inputs encoded by the VAE after training. One can see that structurally similar test inputs are close in latent space.

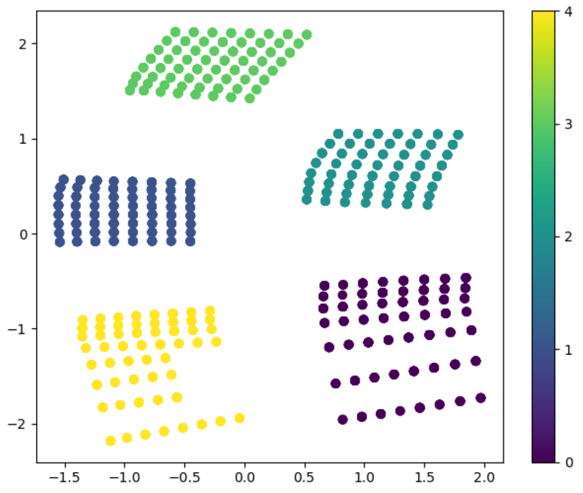


FIGURE 6.9: Derived test inputs in the two-dimensional latent space (spanned by the variables *mean* and *stddev*) of the trained VAE. The different colors indicate the five artificially designed patterns from which test inputs were derived.

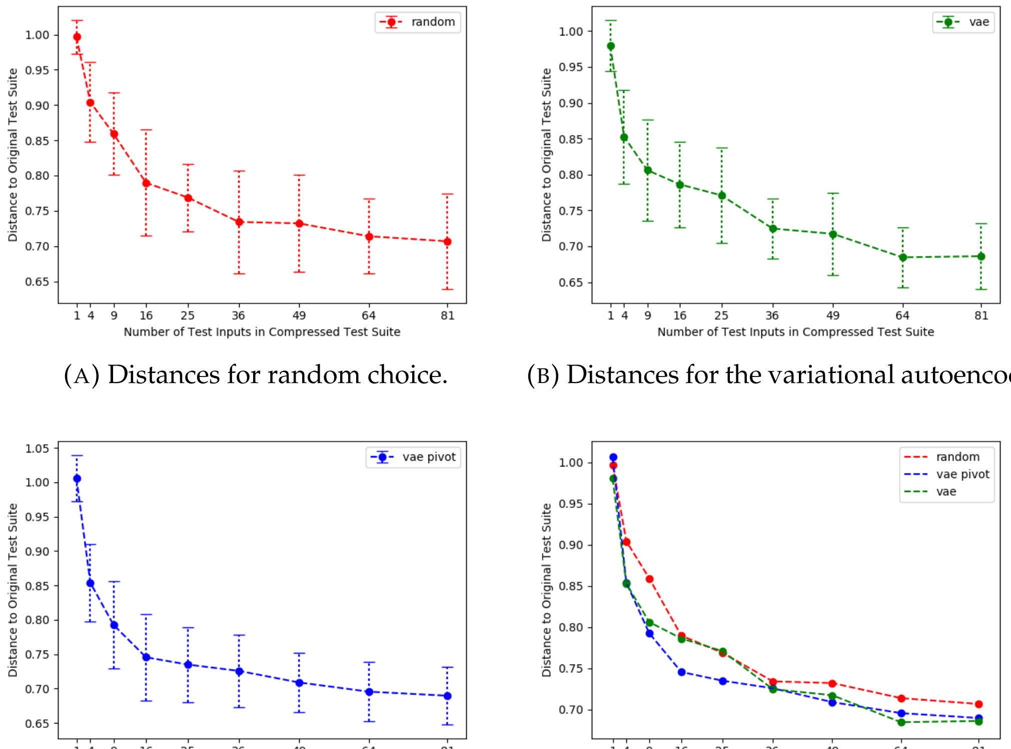
6.3.3 Experiments

A VAE trained with all the test inputs of a uniform test suite can be used to generate arbitrary numbers of inputs again. In terms of the task of compressing the suite, this, on the one hand, includes an essential opportunity in contrast with approaches that just select the required number of inputs from those defined in the original suite: Using the VAE, we primary generate hitherto unseen test inputs. Though these are not included within the original test suite, they reflect its deeply coded structure. Consequently, even more facets of the system behavior can be triggered for testing, without the need for additional human effort.

On the other hand, the generation of hitherto unseen test inputs poses the risk of losing facets covered by the original test suite. This could lead to worse compression coverage (cf. Sect. 6.2) than achievable by approaches selecting existing test inputs.

Nevertheless, in [RK17a] we expected that generating test inputs with the help of the trained VAE at least leads to better compression coverage results than randomly choosing test inputs from the original suite. We investigated this expectation by sampling values from the two-dimensional latent space of the trained VAE which was then used to reconstruct the related test inputs. In order to simplify the sampling process, we exclusively considered square numbers as compression sizes. Given such a size n, we first calculated $\sqrt{n}$ linear spaced points between 0.01 and 0.99. As the VAE interprets the prior of the latent space as being normal distributed, we transformed the resulting points then through the inverse of the cumulative distribution function [Cho16]. The final samples were obtained by constructing the cross product of the resulting points.

Additionally to the mentioned straightforward training of VAEs on test inputs,



(A) Distances for random choice.

(B) Distances for the variational autoencoder.

(C) Distances for the variational autoencoder that considered pivot samples.

(D) Comparison of distances.

FIGURE 6.10: Comparison of compression quality which is assessed by distance Δ^{π}_{TS} of the original and the compressed test suite. The smaller this distance the higher compression quality in term of the compression coverage goal as proposed in Sect. 6.2. Results for the different compression sizes are averaged over 10 independent runs. The error bars depict the standard deviations.

we evaluated another variant: We augmented each test input vector with its behavioral distances (cf. (6.1)) to 10 pivot elements, i.e., test inputs from the original test suite that were randomly selected before training. We expected that considering those exemplary distance measures would help this extended model (we refer to it as Pivot VAE (PVAE)) to learn an even more meaningful latent space whose structure correlates with our behavioral distance measure. Apart from the modified input space we chose the same meta-parameter values as for its above mentioned standard VAE sibling.

To compare compression quality, we executed the processes of (1) training the standard VAE and sampling n test inputs as described above, (2) training the PVAE and sampling n inputs, and (3) randomly choosing n test inputs from the original test suite for all $n \in \{1, 4, 9, 16, 25, 36, 49, 64, 81\}$. Figure 6.10 shows the results averaged over 10 different test suites to compress.

As expected, both autoencoders performed better than the random process for the chosen compression sizes, in particular for the lower ones. For instance, considering compression size $n = 4$, the VAE reached a 6 percent, the PVAE a 8 percent smaller distance. Though the compression gain currently is only moderate, recall that most test inputs generated from the learned latent space of a VAE are new in relation to the original suite. With growing n, the random process catches up and eventually overtakes the autoencoders. This is mainly caused by the increasing chance of sampling appropriate representatives of the defined patterns. The generation processes using the autoencoders cannot use this advantage as they do not refer to the concrete test inputs of the original suite. Moreover, because of reconstruction losses not all of the test inputs from the original test suite might be identically producible from the latent space at all. Further meta-parameter fitting might help.

Compared to the VAE, the PVAE performed slightly better. Also this observation matches with our expectations as we assumed that considering behavioral distance measures in training would help the PVAE to find a more meaningful latent space. This observation encourages further work on integrating the goal function of compression coverage into the approach for enabling an even more adaptive process of uniform test suite compression.

6.4 Related Work

This chapter considered the use of neural networks as behavioral test models for self-adaptive systems – and a machine learning-based test strategy for deriving test inputs based on those.

While, to the best of my knowledge, there are no comparable approaches replacing classical models such as FSMs by neural networks in testing self-adaptive systems, other use cases for neural networks in the software testing process had been yet investigated.

Khoshgoftaar et al. [KPM92] use neural networks for predicting the number of faults to be found in particular software programs and show that they can in fact outperform more traditional regression modeling techniques. Kanmani et al. [Kan+07] come to similar conclusions using neural networks for particularly predicting faults in object-oriented software systems. No self-adaptivity and no strategies to be derived for testing had been considered.

Vanmali et al. presented a concept for using a neural network as kind of automated test oracle [VLK02]. They trained the neural network on test sets of version one of a piece of software and evaluated further versions on this. Similar work was done by Aggarwal et al. in [Agg+04], Mao et al. [Mao+06] and Shahamiri et al. [Sha+11]. Although the trained models for oracles can be considered similar to those we considered here – they are essentially mapping system inputs to expected system outputs – they are not meant as basis for deriving test strategies. The presented approach for deriving test inputs using autoencoders might be a fruitful extension thus.

Anderson et al. [AVMM95] presented experimental results in using neural networks for test suite reduction. In contrast to us they however considered deriving subsets from existing test cases.

6.5 Conclusion

The presented evaluations suggest that neural networks can indeed capture adaptive agent behavior; and they can serve as basis for deriving test strategies.

However, there still are several limitations and questions for future work. One interesting issue might, for instance, be how agents can be imitated that implement a feedback loop? In this case the system under tests output at time step t would (partially) depend on the output at previous time steps $t - n$ for some positive n. To cope with that we could fix a particular agent strategy, i.e., find an influence function estimating agent reactions that are in between the observations; or we could (2) decode the feedback mechanism itself by considering historical behavior within the influence function. Kapitel 7 will revive the first of those options.

Although it turned out that the proposed approach for using VAEs to compress uniform test suites slightly outperforms random test input selection for high compression rates, which seems promising since the VAEs generated previously unseen test inputs as opposed to the random process: Even better results might be obtained when further experimenting with presenting additional data about the SuT's behavior to the VAE. Moreover, further meta-parameter fitting might help. For the presented proof of concept, we exclusively used square numbers as compression sizes in order to simplify sampling. This might be generalized by linking the sampling task with the mathematical problem of packing circles in squares in order to reuse approaches that can be found there [LR02].

Chapter 7

Test Policy Adaptation

Testing can be seen as an optimization problem with two antagonistic goals: confidence increase versus efficiency. The more test cases we execute the higher the confidence but also the higher the costs; still, exhaustive testing is impossible. A good test strategy should consequently be designed in a way that it is able to find faults by revealing failures preferably early.

Let us in this context consider what we called Game of Testing (GoT) between the tester and the SuT again (cf. Sect. 3.3.2). While the tester tries to reveal as many and as serious failures as possible, the SuT seeks to prevent exactly this. Recall that an instance of the GoT, cf. Fig. 7.1, is described by the tuple $(S_{Env}, A_{\mathcal{T}}, A_{\mathcal{S}}, T, R_{\mathcal{T}}, R_{\mathcal{S}})$, where S_{Env} is a set of states, the sets A_* are the actions the tester $(* = \mathcal{T})$ and the SuT $(* = \mathcal{S})$ are able to take, the transition function $T : S_{Env} \times A_{\mathcal{T}} \times A_{\mathcal{S}} \rightarrow PD(S_{Env})$ defines the effect of actions on the state of the environment with $PD(S_{Env})$ denoting the set of discrete probability distributions over S_{Env}, and the reward functions $R_* : S_{Env} \times A_{\mathcal{T}} \times A_{\mathcal{S}} \times S_{Env} \rightarrow \mathbb{R}$ quantify the goal of the tester and the SuT defining their task.

Assuming discrete time steps $t = 1, \ldots, N$, the SuT as well as the tester are sequentially selecting actions according to their respective policy $\pi_*^t : S_{Env} \rightarrow A_*$, which returns the action $a_* \in A_*$ to execute in a state $s_t \in S_{Env}$. At the end of each time step the environmental state s_{t+1} is updated through $T(s_t, a_{\mathcal{T}}, a_{\mathcal{S}})$. The tester as well as the SuT are rewarded by $r_{t+1} = R_*(s_t, a_{\mathcal{T}}, a_{\mathcal{S}}, s_{t+1})$. By improving their policy, both of them attempt to maximize their particular goal, i.e., to maximize the expected sum of discounted rewards $\sum_{k=0}^{N} \gamma^k \cdot r_{t+k+1}$ with $0 \leq \gamma \leq 1$ at runtime.

Considering the tester's perspective on the GoT, formally we cut out an MDP for a given SuT policy $\pi_{\mathcal{S}}$. This MDP is of the form $(S_{Env}, A_{\mathcal{T}}, T_{\mathcal{T}}, R)$, where S_{Env} and $A_{\mathcal{T}}$ can be directly taken from the GoT and the transition function $T_{\mathcal{T}} : S_{Env} \times A_{\mathcal{T}} \rightarrow PD(S_{Env})$ as well as the reward function $R : S_{Env} \times A_{\mathcal{T}} \times S_{Env} \rightarrow \mathbb{R}$ are formed by embedding $\pi_{\mathcal{S}}$ representing the SuT's runtime behavior within the test environment: $T_{\mathcal{T}}(s, a) = T(s, a, \pi_{\mathcal{S}}(s))$ and $R(s, a, s') = R_{\mathcal{T}}(s, a, \pi_{\mathcal{S}}(s), s')$ for $s, s' \in S_{Env}$ and $a \in A_{\mathcal{T}}$.

Previous sections assumed the policy $\hat{\pi}$ of the SuT fixed – the self-adaptive SuT is not supposed to learn anymore at the time we are testing. What, however, if it does? As the policy of a still learning SuT changes over time, things are obviously getting harder for the tester.

This chapter reports about a case study in which we adapted a tester's policy for a learning version of an SVS at test-time [RK18]. As soon as we expected a change in the system's policy, we adapted also the test policy by using the given simulation.

The first section is going to describe the actual GoT instance considered and to discuss possible solutions. After a little recap about the challenges when testing learning, self-adaptive systems Sect. 7.2 presents the approach we used for experiments the results of which are presented in Sect. 7.3.

The reader should note that some parts of this chapter are direct citations from paper [RK18] which are set in context of present work.

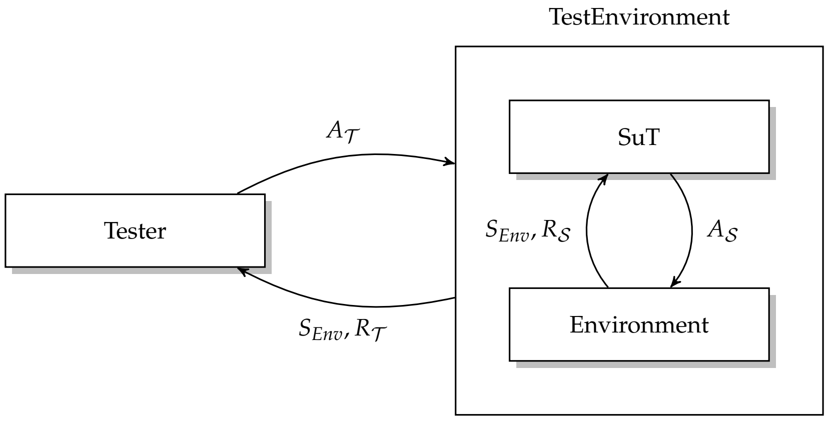


FIGURE 7.1: General setting of the Game of Testing combining two decision processes, one for the SuT and one for the tester.

7.1 Risk-based Testing an SVS

Within the case study we again considered a simulated self-adaptive SVS. It autonomously drives through rooms with obstacles such as staircases, cleaning the dust while avoiding damage arising from collisions or falls. The particular tasks of the SuT could be partly controlled through a mobile device: the user is able to sketch the room the SVS will be located in within the mobile app, and mark positions which should be particularly taken into account for cleaning. Given such a user input the SVS is supposed to immediately drive to the marked position in order to solve the task. Figure 7.2 recalls an exemplary situation.

7.1.1 Game of Testing Instance

A quantification of the traditional testing intent, i.e., finding errors by revealing failures [MSB11], was supposed as test goal. Therefore we extended the binary decision whether a failure $f \in S_{Env}$ is revealed in simulation or not with a weighting function $W : S_{Env} \rightarrow \mathbb{R}$ which can be interpreted as kind of risk prioritization (cf. Sect. 5.1.1). While $W(s)$ is 0 if s is a correct state, for a failure state an estimate of its impact is returned that might be quantified, e.g., by the monetary costs caused by the failure. An example for such a failure state might be the crash of the considered SVS with an obstacle resulting in particular costs. The tester's reward for choosing $a \in A_{\mathcal{T}}$ in $s \in S_{Env}$, through $T_{\mathcal{T}}(s, a)$ resulting in $s' \in S_{Env}$, is then determined by

$$R_{risk}(s, a, s') = W(s') \; . \qquad (7.1)$$

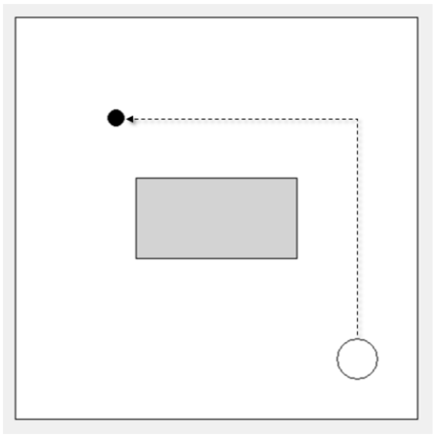


FIGURE 7.2: Smart vacuum system (unfilled circle) within a room which contains an obstacle (light gray rectangle) as well as a user-given mark. The dashed arrow shows a possible path the SVS could take to drive to the marked position.

On the first glance, this risk-based reward lacks the probability of failure occurrence, which is traditionally included in risk considerations [Aml00; Bac99; Rei+16]. This probability is however included implicitly: The more likely a failure is revealed, the more likely its weight will occur in the future rewards of the tester. As the tester strives to maximize the future rewards, the more likely failures have consequently a higher influence on the tester's policy than the more unlikely ones.

For experiments the simulation of the SVS was instrumented with a test interface allowing a tester to modify the SVS's environment, i.e., the room with its features, at runtime. The overall instrumented simulation instantiated the GoT as follows:

- State space S_{Env}: The features of the room comprising its dimension, the position of the SVS, the position and dimension of an included staircase, and possibly the position of a user-given mark.
- Tester's actions $A_{\mathcal{T}}$: Mimicking possible user interactions the tester can place a mark at an arbitrary position of the virtualized room using the app.
- System's actions $A_{\mathcal{S}}$: The SVS can drive in any direction; it is only limited by walls it cannot pass.
- Transitions T: For experiments we assumed a deterministic transition function in simulation. The SVS position changes with the direction it drives in. A mark set by the tester remains until the SVS reaches the marked position. As soon as this is the case, the mark is assumed to be cleared. If the SVS's position is within the staircase, a final state is reached.
- Tester's rewards $R_{\mathcal{T}}$: Following the risk-based testing idea the tester gets rewarded by a weight of 10 if the SVS falls down the stairs, as this is the most risky failure considered here.
- System's rewards $R_{\mathcal{S}}$: The SVS gets a positive reward for reaching entered marks (1 per mark) and a negative reward (-1) for falling down the stairs in order to avoid damage.

7.1.2 Model-based and Model-Free Solution Methods

Let us now consider potential solutions for the tester's MDP. With the reinforcement learning strategies in general Alg. 1, and especially with the Q-learning approach Sect. 5.3.4, I already considered methods for solution. With respect to the tester's MDP the different methods, however, significantly differ in results as the next section will show.

A classification of RL approaches previously suggested by Sutton and Barto [SB98] lets us discuss particular requirements and limitations for the approach to take. According to this the next section will come up with a mixture of both approaches we used in experiments which utilizes elements of the *model-based* as well as of the *model-free* RL methods.

Model-based Methods As the class name suggests, the model-based methods assume an *interpretable* model of the environment, i.e., the state space S_{Env} as well as the transition function $T_{\mathcal{T}}$ including $\pi_{\mathcal{S}}$. This model is used for *planning* future action sequences, e.g., through random search or dynamic programming. For the tester such a method generally means to make predictions about how many and how serious failures might be revealed by possible action sequences on given interpretable models; and to choose that sequence for execution which was predicted to optimize the discounted sum of rewards. For instance, having given a finite state machine abstractly describing the SuT behavior as well as one for its environment, we can use them to search for the most serious failures. As we showed in [Rei+16] the resulting traces of transitions can be translated to actual test cases afterward.

Model-free Methods The model-free methods, on the other hand, can be utilized without an interpretable model for S_{Env} and $T_{\mathcal{T}}$. Letting a software agent continuously interact with an executable environment interface that implements the states and transitions, they successively *learn* to form preferably profitable action sequences. The previously considered Q-learning approach is one of the most commonly used representative of those model-free methods [Wat89]. Recall that as a so-called temporal-difference variant, it lets the learning agent sequentially update an *action-value function* $Q : S_{Env} \times A \to \mathbb{R}$ by exploration at runtime (in our case, the action set A is $A_{\mathcal{T}}$). $Q(s, a)$ is meant to assess the expectation of rewards gained in the future if choosing $a \in A$ in $s \in S_{Env}$.

The concrete action selection at runtime is considered a trade-off between exploration and exploitation. While exploitation of previously gained knowledge which is bootstrapped by the Q-function for the agent means to choose $\operatorname{argmax}_a Q(s, a)$ in all states s, exploration mostly means choosing random actions to find out their previously unknown value.

For a tester using a model-free RL method, such as Q-learning, consequently implies to include random testing for exploration purposes within the policy. This is obviously at the cost of efficiency. Further, there is a second disadvantage when using the model-free methods: the learned policy particularly fits for a single reward function and as this function changes, it is to be learned from the beginning again. This is because those methods directly learn the expectation values for particular decisions. If a tester, for instance, uses Q-learning for solving the GoT and modifies the weight function for failures after a while, he will have to start learning $\pi_{\mathcal{T}}$ from scratch.

Consequently, the model-based methods are generally preferred in software testing. This is also due to the fact that usually abstract and thus manageable models are already available from system design, which can be cheaply leveraged. As I, however, already showed – and will show in the following again – this is not the case for self-adaptive systems. We are forced to use the model-free RL variants for solving the GoT for the tester.

7.2 Testing a Still Learning System

To visualize the particular challenges the case study confronted us with, let me recap the different implementations of a smart vacuum system under test (SVS) I introduced in Sect. 3.1.2. Based on an interface sketching the SVS behavior in an abstract way we will discuss the non-adaptive, the adaptive, and finally the self-adaptive solution I previously suggested. Section 7.2.1 ends up presenting the version we actually faced in present experiments. Section 7.2.2 and Sect. 7.2.3 discuss how we wanted to tackle this challenge which is setting the basis for evaluations described within the following sections.

7.2.1 The Problem with Testing Systems Learning at Test Time

As sketched in Lst. 7.1 an SVS generally interacts with its environment by receiving inputs from built-in sensors, such as infrared and bumping sensors, and by driving actuators, such as an engine and wheels. For the sake of simplicity we assumed that `sensorCtr.sense(D)` returns one numerical value that merges all the sensory inputs which are relevant for driving in direction `D`. Method `plan(inputs[]) : Action` realizes the decision process between the sensors and actuators, mapping the inputs to outputs by actions.

LISTING 7.1: Abstract SVS

```java
public abstract class SVS {
   SensorCtr sensorCtr = new SensorCtr();
   ActuatorCtr actuatorCtr = new ActuatorCtr();

   /** This method controls the SVS behavior */
   public void drive(){
      // Get inputs for all directions
      Map<Direction,Integer> inputs = new HashMap<>();
      inputs.put(Direction.North, sensorCtr.sense(Direction.North));
      inputs.put(Direction.South, sensorCtr.sense(Direction.South));
      inputs.put(Direction.East, sensorCtr.sense(Direction.East));
      inputs.put(Direction.West, sensorCtr.sense(Direction.West));

      // Plan based on inputs
      Action action = this.plan(inputs);

      // Execute chosen action
      actuatorCtr.act(action);
   }

   /** Plan which action to execute */
   public abstract Action plan(Map<Direction,Integer> inputs);
}
```

Non-adaptive Implementation A non-adaptive, transactional implementation introduced by Lst. 2.2, could base the action selection on an evaluation of sensory inputs for each direction. A `Policy` interprets the calculated score, say it deterministically takes the action with the best score. From the point of view of the GoT, the approach the system will take at runtime is thus known to the tester at design time. Further, it is also fixed over time, i.e., $\pi^t_S = \pi_S$ for all time steps t; the code itself provides the model of the MDP.

Given this kind of a SuT the tester is able to plan a strategy a priori. He could for example optimize some code coverage criteria, such as statement or branch coverage, by selecting an appropriate set of sensory inputs. Also a risk prioritization of inputs seems rather obvious. Since the tester is able to *interpret* the system behavior *before* execution he is able to make use of the model-based solution methods.

Self-adaptive Implementation Considering the adaptive implementation depicted in Lst. 3.1 we saw that the introduced ability of runtime adaptation would in principle not break the interpretability for the tester. If there are multiple strategies but the strategy choice of the SuT was static and/or fully controllable by the tester, we still would be able to utilize model-based solution methods for the tester's MDP.

By introducing the reasoner in Lst. 3.2 we let the adaption mechanism, however, also depend on a knowledge base whose state is unknown to the tester. This is what we referred to as a *self-adaptive* implementation of the SVS.

Previous sections considered the case in which a self-adaptive SuT will not continue its training at test (and execution) time. It still holds that $\pi^t_S = \pi_S$ for all time steps t while testing. In this case the suggestion was to expand the classical process of testing by learning the actual SuT behavior and capturing it in some test model again (see Sect. 3.3). After that learning phase we could thus apply some model-based techniques again.

Implementation without Predefined Strategies Listing 7.2 sketches the more direct implementation of self-adaptivity than this depicted in Lst. 3.2 we used for experiments I am going to report about here. Instead of choosing one of a given set of strategies we let the reasoner directly work on the set of possible actions.

Further, the experiments considered the case in which the SuT is going on learning, even at test-time, which invalidates the previously considered process in which a model for testing was learned a single time. We were forced to use a model-free variant for solving the tester's MDP.

To set it in context with the GoT assume there is a natural number n for each time step t for which $\hat{\pi}_t \neq \hat{\pi}_{t+n}$. Since the system's policy is part of the tester's environment we can see that a previously optimal test policy at t might be invalidated at $t + n$ as the tester's MDP changes. To get it optimal again the tester needs to react and to adjust it at test time as well.

The SVS's knowledge base was implemented by a neural network which is continuously updated at runtime using the *Deep Q-Learning* (DQL) algorithm, a variant of the classical Q-learning method (cf. Sect. 7.1.2; the interested reader may want to refer to [Mni+15] for a detailed explanation of this algorithm).

LISTING 7.2: Self-adaptive Implementation of SVS

```
package svsControl;
import java.util.Map;

public class SelfAdaptiveSVSv2 extends SVS {
  private Reasoner reasoner;

  public SelfAdaptiveSVSv2(Reasoner reasoner) {
     this.reasoner = reasoner;
  }

  @Override
  public Action plan(Map<Direction,Integer> inputs) {
     reasoner.updateKnowledgeBase(inputs);
     return reasoner.chooseAction();
  }

  interface Reasoner{
     public void updateKnowledgeBase(Map<Direction,Integer> inputs);
     public Action chooseAction();
  }
}
```

7.2.2 Direct Future Prediction

Aiming to profit from the advantages of the model-based RL approaches, but forced to use a model-free variant, in our experiments we came up with a mixture between both for continuously solving the tester's MDP. Instead of using a given model of the environment, just as the model-based methods do, we argued for learning and sequentially adapting a test model at runtime; and instead of directly learning the expected value of particular decisions, just as the model-free methods, we argued for planning at runtime in consideration of the current test model state.

This approach we proposed for solving the tester's MDP was strongly inspired by findings of Dosovitskiy and Koltun [DK17]. Originally classified as an approach to sensorimotor control in immersive environments, their *Direct Future Prediction* (DFP) can be generally used for autonomously solving MDPs. Doing so it constitutes a mixture between the model-based and the model-free RL methods.

Similar to related approaches [LS02; Sin+03], the general idea of DFP is to train a *predictor* P, i.e., a model for the function $S \times A \to S^{Tmp}$, mapping a state $s \in S$ (in case of the tester's MDP S_{Env}) and an action $a \in A$ (in case of the tester's MDP $A_{\mathcal{T}}$) to a number of expected future states at given temporal offsets Tmp. Since predicting high-dimensional states is still infeasible, the DFP approach concentrates on so-called measurements, i.e., a (possibly processed) subset of state features which is relevant for determining the goal achievement. An observation o_t at time step t is thus supposed to encapsulate the observed state s_t itself and a measurement vector m_t. A *goal* is represented by a function $u(f;g)$ where f is the vector of future measurements $(m_{t+\tau_1}, \ldots, m_{t+\tau_n})$ for $\tau_1, \ldots, \tau_n \in Tmp$ and g are arbitrary goal parameters. As the goal parameters g influence planning and thus the overall progress, the predictor takes them as input. In total this leads to the model signature $P : O \times A \times G \to M^{Tmp}$ where O is the set of observations each comprising a state $s \in S$ and a measurement $m \in M$ and G is the set of goal parameters.

If the predictor P is implemented by a function approximator, such as a neural network, a *prediction* p_t at time step t, i.e., a number of future measurements, is determined by $P(o_t, a, g, \theta)$, where θ represents the learned parameters of the function approximator. This function approximator can be trained by any supervised learning technique with training examples of the form (o, a, g, f) with o, a, g as inputs and f as output. Following Dosovitskiy and Koltun [DK17] we utilized for our experiments a back-propagation approach using a regression loss:

$$L(\theta) = \sum_{i=1}^{N} \|P(o_i, a_i, g_i, \theta) - f_i\|^2 .$$

Except the goal parameters g_i which are internally given to the learning agent, the set $D = \{(o_i, a_i, g_i, f_i)\}_{i=1}^{N}$ of training samples is collected at runtime by sequentially interacting with the test environment, similar as it is depicted in Fig. 7.1: the measurements, the one included in o_i as well as those of f_i are extracted by the agent from the returned states, where collecting all the future measurements usually requires multiple interactions. An implementation of our DFP variant can be found at GitHub[1].

Instantiation In order to balance exploration and exploitation, we used an ϵ-greedy runtime policy $\pi_{\mathcal{T}}$, which means that for an $\epsilon \in [0, 1]$ the best known action a (assessed by the predictor) in state s is chosen with probability $1 - \epsilon$ and a random action otherwise. Assuming that less and less exploration is needed over time, the ϵ is linearly decreased from a predefined maximum $\epsilon_{\max} \in [0, 1]$ to a minimum $\epsilon_{\min} \in [0, \epsilon_{\max}]$ within a fixed number of time steps.

Applying this concept for testing a particular SuT includes the following engineering decisions:

- *Temporal offsets*: The number and value of temporal offsets determines the horizon of the learner. The nearer this horizon, i.e., the less steps to be viewed in the future, the easier the predictions but the more short-sighted the planning. This choice is considered highly domain-specific.
- *Measurements and goal function*: Since the (goal) function u uses future measurements for goal assessment, its implementation needs to go hand in hand with defining the portion of state features to be used as measurements. In order to emulate the risk-based reward function proposed before, such measurements need to be found that signal the occurrence of failures.

7.2.3 A Modified Test Process

As can be seen in Lst. 3.2 the state of the SVS's world model can generally change in every time step. We are, however, neither able to predict its future states nor do we want to start a separate test process in every time step again, as this would boost the costs.

Thus, by way of compromise we chose the development and testing process depicted in Fig. 7.3 for experiments. It alternates between world-model adaptation and testing phases at runtime as follows:

[1] `https://github.com/dremonaut/action-provider-rl`

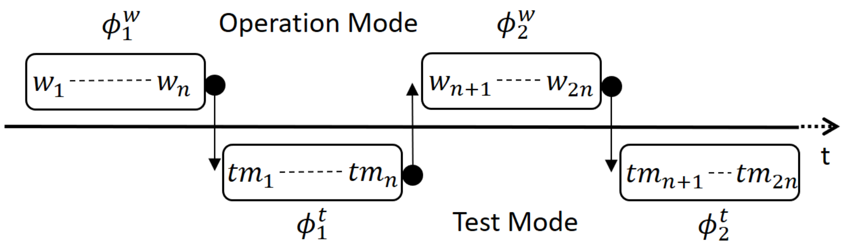


FIGURE 7.3: The alternating operation and test process we considered in experiments.

1. In operation mode, i.e., the phases ϕ_1^w and ϕ_2^w in the figure, the SuT follows its policy and sequentially updates its world-model state w_t by means of self-adaptation, just as we described before.
2. At specific points in time (let us, for the time being, assume a fixed time interval) the test mode is started: In phases ϕ_1^t and ϕ_2^t a tester, such as the DFP and the DQL test agent, tests the SuT in its current world-model state, w_n and w_{2n} respectively, continuously refining its test-model state, starting with tm_1 and tm_{n+1} respectively, at runtime for maximizing the reward.
3. Steps 1 and 2 are sequentially repeated until the confidence in the SuT behavior is considered sufficient. The operation and the test mode might also run concurrently in separate simulation instances.

While we not investigated them further in experiments, we envisioned the following two extensions for the presented testing process, both motivated from classic regression testing [MSB11]:

First, the failures detected within a testing phase could be directly used for a kind of re-adaptation of the SuT's world model. For the learning SVS this would just mean to get presented the found failure scenarios in order to learn avoiding them. This resembles fixing a revealed bug by a developer. Second, one could correlate the length of the testing phases with the number of failures previously revealed. If no failure is found in a phase, the length of the next testing phase might be reduced.

7.3 Adapting a Test Strategy at Runtime

The intent of our experiment was investigating whether the DFP tester agent establishes a balance between confidence and efficiency mentioned in the introduction by comparing it against other variants.

While we assumed the efficiency goal directly measurable in our process by means of the number of time steps consumed by the testing phases, we assumed the confidence in the SuT's behavior depending on the failure detection capability of the tester applied. The faster the tester (agent) is usually able to reveal possibly many failures the faster we gain confidence that all existing errors were detected.

7.3.1 Approaches to Compare

While the knowledge base of the SuT determining its strategy for optimizing $R_{\mathcal{S}}$ was sequentially adapted by training, we compared the following methods for automatically solving the tester's MDP:

DFP Tester The DFP tester agent was implemented just as described in Sect. 7.2.2. As measurements we chose the two values of the SVS's x and y position within the room. To emulate $R_{\mathcal{T}}$, the used goal function $u(f;g)$ evaluates whether the predicted SVS coordinates in f are within an obstacle's area, which is passed as parameters g. In this case the same reward as $R_{\mathcal{T}}$ is distributed. For future measurements we used temporal offsets of $\tau = 1$, $\tau = 3$, and $\tau = 5$. Apart from the inputs and outputs, the neural model for the predictor was implemented by 3 fully connected hidden layers with 256, 128, and 64 neurons, respectively. This setup of future measurements and neural model, which comprises 48 710 parameters, resulted from an informal trial and error search. For runtime action selection we chose an ϵ-greedy policy with linearly decreasing ϵ as described in Sect. 7.2.2 with $\epsilon_{\text{max}} = 1$ going down to $\epsilon_{\text{min}} = 0.05$ in 5000 test steps.

DQL Tester As representative for the model-free variants we implemented a second tester agent which, similarly to the SVS itself, used a DQL implementation for solving the GoT. To ensure comparability with the DFP tester agent, we modified the classical DQL approach presented in [Mni+15] by shifting the signature of the Q-function model from the state-centric view $S \to \mathbb{R}^A$ to a state-action view $S \times A \to \mathbb{R}$. For the neural network we applied the same parametrization as for the model of the DFP tester agent, except for different input and output types. This led to 44 033 parameters in total. Also the same policy for runtime action selection was used.

Random Tester Random testing was mimicked by a DFP tester agent using an ϵ-greedy policy with ϵ constantly kept at 1.

7.3.2 Hypotheses

It is usual to estimate the failure detection capability by the number of intentionally introduced errors that the test method to be assessed reveals [DLS78; Rot+02]. Introducing errors in an autonomous system is, however, rather not trivial. We thus used a slightly different estimate instead: Testing the learning SVS in an early stage of training, we assumed that there are enough failures reachable for comparing the detection capabilities of the three different methods to evaluate.

The question was thus if the DFP tester agent finds existing failures earlier than the others. If this is the case, it is more efficient still reaching the same confidence. Revealing more failures than the compared methods within the same time slot would additionally suggest that the DFP tester agent actually justifies a higher confidence.

Those ideas led us to hypotheses (H1) bis (H4) which we pursued in the evaluation:

H1. The DFP tester agent performs at least as well as the DQL tester agent when striving for a single goal. Both methods obtain good risk scores w.r.t. the presented reward function, at least they are better than a base line constituted by random testing. If this is the case it is worth adapting the test model at runtime by RL.

H2. If we change the goal at runtime, e.g., by introducing new failures to be considered, the DQL tester agent needs to start learning from scratch while the DFP tester agent is able to generalize. If this is the case it is worth employing the DFP tester agent instead of the DQL tester agent.

H3. By using a function approximator as predictor we observe a generalization effect over the test phases. This implies that the test model learned w.r.t. a particular world-model state of the SuT can be also used to test the SuT in another particular state. If this is the case, we can boost confidence as well as efficiency in testing by employing the same DFP tester agent for all the testing phases.

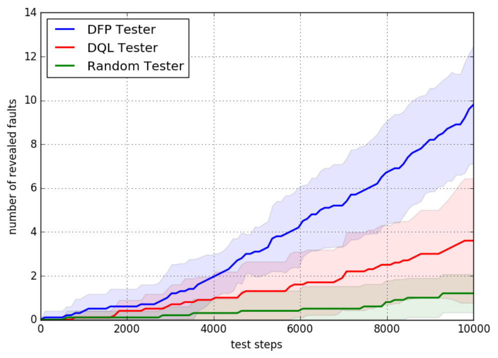
H4. The failures revealed by the tester agents differ from one another – they are diverse. If this would not be the case, it would at least be questionable if the increasing number of revealed failures justifies an increase in confidence.

7.3.3 Experimental Results

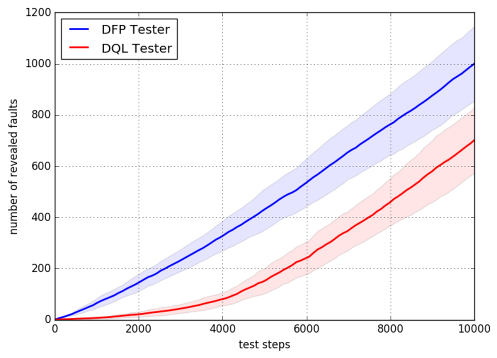
For evaluating the mentioned hypotheses we considered 10 instances per testing method respectively. Further, we considered different room types with different staircase dimensions as source states while the simulation was reset each time the SVS fell down the stairs, the position of an input mark was reached, or after a maximum number of 10 time steps.

(H1) First, we compared the failure detection capability of the different methods within a single testing phase, say ϕ_1^t. The simulation included the SVS world-model state after a training of 500 000 time steps in ϕ_1^w. Figure 7.4a shows the results for 10 000 time steps of ϕ_1^t. It turns out that both of the RL methods, the DFP as well as the DQN tester agent, outperform random testing. Further, we see that the DFP tester agent on average finds more failures than the DQN tester agent. This confirms hypothesis (H1) and uncovers another advantage of the DFP over the DQN: As the DFP not directly learns particular action values, but rather an abstract model of the environment, it is not as sensitive to sparse rewards as the DQN. This fact explains the far better performance of the DFP tester agent.

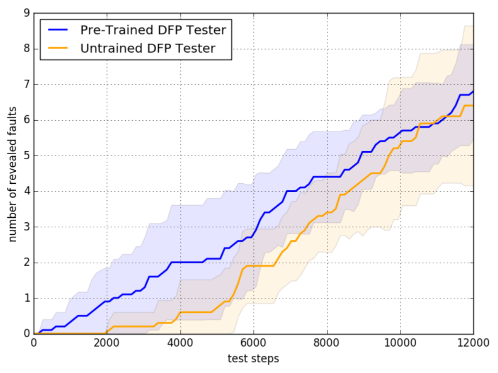
(H2) After investigating (H1) we modified the test goal: instead of rewarding the tester for attracting the SVS to fall down the stairs we defined another obstacle within the room placed at a fixed position, striving to test if the SVS is able to crash with it. Instead of training the tester agents from scratch we reused the pre-trained models from the evaluation of (H1). Figure 7.4b shows the results. They suggest that the DFP tester adapts much faster to the modified goal. This conforms with hypothesis (H2), as it suggests that the model-free RL variant, in contrast to the DFP, needs to start learning the test strategy from scratch once the test goal is modified.



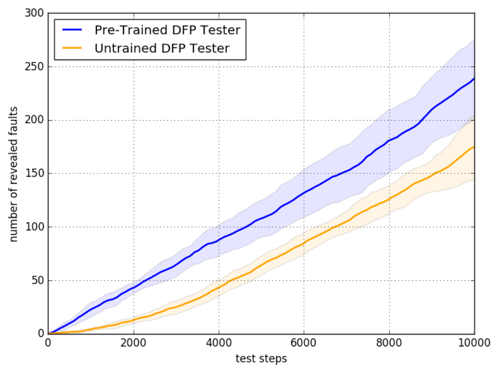
(A) Results for (H1) showing the performance of the three tester agents within a single test phase.



(B) Results for (H2) showing the generalization capability of the tester agents over different goals.



(C) Results for (H3) showing the generalization capability of the tester agents over the test phases.



(D) Results for (H3) showing the generalization capability of the tester agents over the test phases and different goals.

FIGURE 7.4: Experimental results for evaluating hypotheses (H1), (H2), and (H3). The plots show the mean results gained by 10 tester agent instances respectively. The colored area indicates the spread in results by means of the standard deviation.

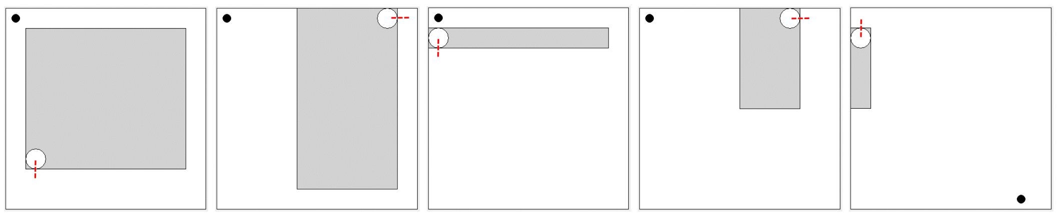


FIGURE 7.5: Situations in which the SVS fell down the stairs. The red dashed line indicates the last SVS action before the fall.

(H3) To measure the inter-phase performance of a DFP tester agent, i.e., its failure detection capability in ϕ_2^t after being trained in ϕ_1^t, we continued training the SVS world model in ϕ_2^w for another 500 000 steps. Again we reused the 10 DFP test models from evaluating (H1). Figures 7.4c und 7.4d compare the performance of those pre-trained agents with the performance of 10 new DFP tester agent instances which had to learn their models from scratch w.r.t. both of the rewards: the one considered for evaluating (H1) and the one for (H2). The results suggests that the insights our tester gained about the environment in an earlier testing phase accelerate training in the following. We can thus assume that there is a kind of generalization that increases the failure detection capability of the DFP tester agent raising its efficiency over the test phases.

(H4) Assessing the diversity of failures revealed, Fig. 7.5 plots a randomly chosen subset of situations in which the SVS erroneously fell down the stairs. Even if there is a visible pattern in those situations they are not equal by far. In fact, identifying such patterns in revealed failures practically helps us to localize and fix the underlying bug. In the particular case considered here, in which the SVS is learning its world model from simulation, simply presenting those failures might already fix them.

On the whole, the experiment confirmed our hypotheses. Doing so, and having another advantage of the DFP over the DQN found by evaluating (H1), this indicates that the DFP tester agent is able to autonomously and efficiently increase the confidence in a SuT – more than a comparable method using DQL and more than random testing. Combined with the high failure detection capability, the high automation degree makes testing self-adaptive systems at runtime feasible.

7.4 Related Work

We formulated testing self-adaptive systems at runtime as a game, the Game of Testing, and proposed a Reinforcement Learning-based method for solution. There were similar formulations for testing tasks before [Yan04; Bla+05; Nac+04]. Also RL has been used for solving test issues, such as optimizing code coverage [Gro11; VRC06] or prioritizing test cases in Continuous Integration [Spi+17]. All those approaches did, however, not consider the particular challenges for testing self-adaptive systems.

The research field of *Adversarial Machine Learning* considers similar games as the GoT for training possibly robust classifiers [LM05]. Goodfellow et al. [Goo+14] proposed a framework for estimating a generative model by playing a game against a discriminative model that estimates if a sample comes from training data or from the generative model. Though the idea behind is quite similar to ours, their proposed solution is specifically designed for classification tasks and cannot be used for solving arbitrary GoTs. Pinto et al. [Pin+17] proposed applying an adversarial agent, next to the actual one, in reinforcement learning tasks which applies disturbance forces to the environment in order to gain robustness for the actual agent. The game between the both is formulated as a zero-sum game. This can, in fact, be viewed as an instance of the GoT solved by RL. The approach discussed here is, however, more general: First, we consider self-adaptive systems in a whole instead of RL agents in particular. Second, our method is not limited to zero-sum, min-max objectives; in fact, the goals we considered in experiments are none of both. Third, they used – in contrast to us – a model-free method for solution, suffering from all the disadvantages discussed in Sect. 7.1.2 if used for our GoTs.

Following the discussion in Sect. 7.1.2, other approaches proposed for testing self-adaptive systems can be categorized by the fact whether they use a model for testing. As an instance of the first, and in conformance with the models@runtime community [Che+14], Hänsel et al. [HVG15] proposed the use of runtime models for testing self-adaptive systems. This relies on the assumption that such runtime models representing the adaptable software and the environment are, in fact, given, which, however, is not always the case. The model-free approaches, including the evolutionary approaches proposed by Nguyen et al. [Ngu+12] and Fredericks et al. [FDC14], on the other hand, still suffer from the problems we described in Sect. 7.1.2. In general, we see our GoT as an adequate framework for classification and comparison of existing and future approaches on testing self-adaptive systems.

As testing is shifted to runtime, the discussed experiment has similarities to the area of runtime verification. The basic process there is (1) creating a monitor that is able to discriminate erroneous from correct system traces, (2) instrumenting the SuT for obtaining the traces, (3) running the system, and (4) verifying the received event traces [FHR13]. As opposed to our ideas, this process does not consider selecting appropriate test inputs at runtime, but rather concentrates on given program traces. Generalization, as we supply over observed traces and given goals, is thus rather not needed for planning there.

7.5 Conclusion

This section presented a method for runtime testing self-adaptive systems. Building on the GoT the suitability of model-based and model-free *Reinforcement Learning* variants was discussed for solution, with the help of a concrete example. It turned out that the model-based approaches fit the bill, but, at least in their pure form, cannot be utilized for the kind of SuTs we consider. This is because self-adaptive systems lack interpretable design time models. On the other hand, the model-free variants such as Q-Learning have turned out to be too specialized on particular reward functions for testing; further, the sparse reward problem restricts their failure detection capability.

By way of compromise the use of *Direct Future Prediction* as a mixture between the both reinforcement learning variants has been proposed. It works similar to the model-based methods, but learns the (test) model at runtime instead of assuming it given beforehand. As experiments showed, this technique is able to generalize across different test goals and adaptation steps of the SuT at runtime and suffers far less from the sparse reward problem than Q-Learning.

But there is another advantage we did not make use of yet: In contrast to the model-free methods the DFP is generally able to cope with history-dependent reward functions. This capability opens the door for coverage-based test goals which, for assessing the value of future actions, need to take into account the previously executed ones. Further test goals that should be investigated in the GoT context include mutation-based as well as non-functional test goals. Through finding some kind of design patterns for solving the meta-parameters within the measurement selection, architecture and goal function for the predictor, I envision to establish even more efficiency in maximizing the confidence in self-adaptive systems by testing.

Chapter 8

Adaptive Test Policy Execution

Recall the modified test process described in Sect. 3.3. Test analysis and design is done on the basis of an inferred model of the self-adaptive SuT. The result is a concrete test strategy comprising test cases for implementation and execution in reality, i.e., a not simulated but realistic environment.

The technique described in the chapter before for example lets us derive test strategies for a self-adaptive SVS. By returning tester's actions for given environmental states it is defining test inputs for the SuT in given situations. In this chapter I want to consider applying such derived inputs, i.e., environmental states in reality. Therefore I am going to report about an experiment (see [RK17b]) we did considering an SVS as SuT again. As we found the application of inputs derived from simulation in reality implies yet other challenges: the environment of the SVS (i.e., the room) will not be fully *controllable* for the tester as it might be in simulation. Not all of the environmental states can be established in reality, because of, e.g., organizational or financial restrictions. For instance, the dimensions of a room that is provided for testing will be fixed and cannot be changed arbitrarily. This results in limitations for test design.

In order to illustrate these limitations, let us consider a simplified view on the game of testing the SVS: (1) We directly search for a set of *test inputs* in form of interesting room setups by exploring a virtual, simulated world possibly using automatable techniques optimizing particular coverage metrics in order to (2) reconstruct these later within a real environment to compare the actual with the intended system behavior. As discussed and shown before, it is quite common to base test design on environmental and behavioral models (here the simulation) of the SuT. However, as we are not able to set up arbitrary rooms in (2), we have to restrict the search in (1) to those we can construct. This seems like wasting the power of simulation, as several insights that could be gained within the rejected scenarios might have been useful for further testing.

To face this limitation, we need a way to generalize insights gained in (1) within arbitrary rooms, in order to instantiate them later to concrete setups in (2). In other words, methods are needed to transfer test inputs between different, uncontrollable setups in a way that retains the intention with which the test inputs were originally designed.

Our experiment was based on the assumption that the intention of applying a test input is to trigger particular behavior of the SuT— or rather of its digital twin (as we called the simulated version of the SuT). This assumption seems reasonable, since many viewpoints in testing, such as the risk-based view or the coverage-oriented view, are based on expectations concerning the system behavior.

The proposed method (see Sect. 8.1) builds on a distance function Δ which compares environmental states $s = \langle u, c \rangle$ comprising uncontrollable as well as controllable features. In the example of the SVS, the uncontrollable part u of the environment state could be represented by particular room setups; and the controllable part c could be shaped by placing a mark at a desired position. The distance function utilizes a notion of *diversity* of expected resulting system behavior which measures the difference of particular actions the system is presumed to take. These are, in the SVS example, driving north, south, west, or east.

The action difference can be arbitrarily quantified, though in experiments we evaluated a single useful concrete instance. Transferring a test input which applies the controllable feature assignment c within the uncontrollable setup u to another setup u' then means to find an assignment c' that minimizes $\Delta(\langle u, c \rangle, \langle u', c' \rangle)$. An algorithm is presented which uses neural models for learning our metric by simply observing the SVS (in simulation or in reality), without having any structural insight (see Sect. 8.2). As we will see first experiments showed promising results (see Sect. 8.3). By simply observing the SVS in simulation, the learned distance measure actually resulted in test inputs triggering similar behavior patterns over different room setups.

The reader should note that some parts of this chapter are direct citations from paper [RK17b] which are set in context of present work.

8.1 Distance-based Test Input Transfer

The experiments considered the following situation: We, as test engineers, aim to validate that system *Sys* behaves correctly. Suppose that the environmental state space S (including its controllable as well as its uncontrollable portion), the *Sys*'s action space A, and an environmental *transition function* T is known. For simplicity, let T initially model the consequence of executing action $a \in A$ in state $s \in S$ deterministically by mapping this pair to a particular successor state, i.e., $T : S \times A \to S$. Let us, for the moment, furthermore assume that we were able to formalize the intended behavior of *Sys* within a deterministic policy $\pi : S \to A$ which maps each $s \in S$ to that $a \in A$ which *Sys* would choose.

8.1.1 Non-producible Test Inputs

Suppose that, pursuing the test objective of gaining confidence that *Sys*, indeed, implements π, we have derived a suite of environmental states $TS \subseteq S$ with which we would like to confront *Sys*. In fact, in experiments we understood $t \in TS$ as test inputs, masking additional properties of proper test cases, like pre- and post-condition or a test oracle and, in the context of the Game of Testing, already including the effect of tester's actions to the environment. Confronted with such an input $t \in TS$, the *expected* behavior of *Sys* is the action sequence $\pi(t), \pi(T(t, \pi(t))), \ldots$. If we want to observe the *actual* behavior, we, however, still have to establish t within a real environment of *Sys* — and this might be practically impossible because of the uncontrollable features within the environmental state space. Consequently, the question arises on how to convince ourselves of the system's correct behavior, though we cannot investigate its behavior for all of the previously defined test inputs.

Let $P \subseteq S$ be the set of all *producible* states, i.e., the states that can be established within the real *Sys* environment. Then, the challenge is to handle the non-producible test inputs in $TS \setminus P$. We claimed that simply ignoring these previously derived test inputs reduces the overall confidence in the correct behavior of *Sys*. In order to keep the test plan stable, we proposed instead to replace each $t \in TS \setminus P$ with another one $\hat{t} \in P$ to which *Sys* is supposed to answer in a *similar* way (w.r.t. π). The assumption behind this was that the process of deriving T was based on intended system behavior, e.g., by using behavior-based coverage or risk-based metrics.

Consequently, determining the right substitute for a $t \in TS \setminus P$ demands to compare the intended system behavior there with that in each $\hat{t} \in P$.

In [RK17b] we provided a recursive definition of this kind of behavior-based comparison by a distance metric $\Delta^\pi : S \times S \to \mathbb{R}_{\geq 0}$ on environmental states. Depending on the policy π, a $t \in TS \setminus P$ should then be replaced by $\arg\min\{\Delta^\pi(t, \hat{t}) \mid \hat{t} \in P\}$. By abuse of mathematical terminology we called Δ^π also a *metric*.

8.1.2 A Policy-based Distance Metric

The metric is built on a freely designable function $\delta : S \times S \to \mathbb{R}_{\geq 0}$ that estimates the one-step distance of two states. As we wanted to focus on the intended system behavior, we initially resolved this by a distance function $\delta^a : A \times A \to \mathbb{R}_{\geq 0}$ on the outgoing actions $\pi(s)$ and $\pi(\hat{s})$, that is $\delta(s, \hat{s}) = \delta^a(\pi(s), \pi(\hat{s}))$. For the time being, let δ return 0 if and only if s and $\hat{s}$ are not final states (this can be included in δ^a by introducing a *void* action type) and the actions are equal; and 1 otherwise.

Based on δ, our distance metric is defined by summing up the state differences along π; we weight the discounted future by the factor $\gamma \in [0, 1[$ to cope with infinite paths:

$$\begin{aligned} &\Delta^\pi(s_1, s_2) \\ &= \textstyle\sum_{0 \leq n} (1-\gamma)\gamma^n \delta(s_1^n, s_2^n) \\ &= (1-\gamma)\delta(s_1^0, s_2^0) + \gamma \Delta^\pi(s_1^1, s_2^1)\,, \end{aligned} \tag{8.1}$$

where $s_i^0 = s_i$ and $s_i^{k+1} = T(s_i^k, \pi(s_i^k))$.

Determinism of the environmental transitions T is a rather strong assumption which, in fact, would not be satisfied for the SVS example. Let us thus move to a probabilistic T. Now, $T(s, a)$ yields the probability distribution of environmental successor states, and we write $T(s' \mid s, a)$ for the probability that T moves from state s to state s' by action a. Our definition of Δ^π accommodates this extension to the probabilistic case as follows:

$$
\begin{aligned}
&\Delta^{\pi}(s_1, s_2) \qquad\qquad (8.2)\\
&= \mathrm{E}\Big\{ \textstyle\sum_{0 \le n} (1-\gamma)\gamma^n \delta(s_1^{t+n}, s_2^{t+n}) \mid s_i^t = s_i \Big\} \\
&= \mathrm{E}\Big\{ (1-\gamma)\delta(s_1^t, s_2^t) + \gamma \Delta^{\pi}(s_1^{t+1}, s_2^{t+1}) \mid s_i^t = s_i \Big\} \\
&= \textstyle\sum_{(s_1', s_2')} T(s_1' \mid s_1, a_1) \cdot T(s_2' \mid s_2, a_2) \cdot \\
&\qquad\qquad \big((1-\gamma)\delta^a(a_1, a_2) + \gamma \Delta^{\pi}(s_1', s_2') \big) \; .
\end{aligned}
$$

Generally, Δ^{π} can be computed for all the state pairs of $S \times S$ by finding an equilibrium for (8.2). Given the policy π, this could be formulated by $|S \times S|$ linear equations in $|S \times S|$ unknowns. An iterative solution for computing the equilibrium inspired by the Bellman equations is therefore advantageous and simultaneously provides us with convergence properties [Bel54]: We calculate a sequence $\Delta_0, \Delta_1, \ldots$ with $\Delta_k : S \times S \to \mathbb{R}_{\ge 0}$ where Δ_0 is arbitrarily chosen and each successor is obtained by the following update rule:

$$
\begin{aligned}
&\Delta_{k+1}(s_1, s_2) \qquad\qquad (8.3)\\
&= \mathrm{E}\Big\{ (1-\gamma)\delta_{t+1} + \gamma \Delta_k(s_1^{t+1}, s_2^{t+1}) \mid s_i^t = s_i \Big\} \\
&= \textstyle\sum_{(s_1', s_2')} T(s_1' \mid s_1, a_1) \cdot T(s_2' \mid s_2, a_2) \cdot \\
&\qquad\qquad \big((1-\gamma)\delta^a(a_1, a_2) + \gamma \Delta_k(s_1', s_2') \big) \; .
\end{aligned}
$$

Then Δ^{π} is a fixed point of this update rule. The sequence $\{\Delta_k\}_{0 \le k}$ converges to Δ^{π} with $k \to \infty$ (similar to the Bellman equation for optimal policies). Using this update rule, the implementation of an algorithm searching for Δ^{π} is rather straightforward. It could be inspired by standard techniques of dynamic programming for solving Bellman equations [SB98].

However, challenges arise if the transition function with the corresponding transition probabilities is not known. The only thing that we can do then is extending the mentioned update rule for estimating T by random experiments. I will show how this can be done, estimating Δ^{π} from observations within a simulation of *Sys* and its environment were T as well as π are unknown.

8.2 A More Realistic Scenario

Let us in the following focus on the case where no π is given, but instead we have access to an executable simulation of the SuT within its environment. As discussed before such a simulation is often accessible for autonomous/self-adaptive systems Sect. 4.1.4. For instance, it could have been used by the system itself for reasoning and learning about appropriate behavior – so that the behavior within the real environment should be fixed.

8.2.1 Considering a Simulation

The main difference of considering a simulation instead of a known policy π is the changing view on the environment as well as on the system's actions. While π enabled us to observe the environment from the system's eyes (as the state and action space the system is intended to base its behavior on were given), the simulation now forces us into a third person view. Exclusively able to observe the concrete state space depicted by the simulation we are deprived of knowledge about the (intended) system's state abstractions. Moreover, we are forced to reason about an artificial space of observable actions the system is able to choose.

Although one will intuitively understand a system action as a kind of function that is invoked within the SuT, we will not be able to monitor all the function executions from the third person view (as traditionally considered in *black-box testing*). By simply observing state changes within a simulation we will notice rather higher level state transitions which were initiated by the system than the lower level building blocks of those. To give an instance, think about the SVS driving one meter northward. There will be a bunch of internal processes for evoking this observable behavior. As work on action abstraction can be, for instance, found in the area of reinforcement learning [SB98], we assume for the sake of this work an appropriate action space to be given.

8.2.2 Learning the Distance Metric using Neural Nets

However, though given such an action space, the distance function still suffers from the possibly very huge state space spanned by all the measurable features of the simulated environment. Typically, these features will even pose continuous parameters, leading to an infinite space of possible environmental states. In consequence, applying the update rule depicted in (8.3) will never reach the equilibrium described in (8.2). In order to, at least, approximate our distance function in a possibly efficient way, in [RK17b] we proposed the use of generalization techniques.

For this purpose, we applied inspiration from reinforcement learning domains, again. Suffering from similar challenges as ours, generalization was successfully integrated there within the last years. Riedmiller used neural networks, the so called Q-networks, to estimate value functions for reinforcement learning tasks [Rie05]. He tackled the divergence issues, which harmed the use of non-linear function approximators for this task before, by repeatedly performing batch updates with entire training data instead of single training examples, as done before, on the Q-network's parameters. Mnih et al. proposed to scale this idea to large data sets by using stochastic gradient updates on mini-batches instead of repeatedly considering the entire, constantly increasing batch [**mnih2015human-short**]. They additionally found that using a separate network for generating target values needed for Q-learning updates improves stability in learning furthermore.

Since the training tasks are similar, both techniques — performing stochastic gradient descent on mini-batches as well as using separate networks for determining the targets when applying the update rule — seem useful for learning the distance function. Thus, Alg. 5 proposes their application to estimate Δ (regarding its general definition in (8.2)) within the *deep behavioral distance learning algorithm (DBDL)*.

Algorithm 5 Deep Behavioral Distance Learning (DBDL): An algorithm for learning Δ^{π} from simulation.

```
Require: $\Delta\big((s_1, s_2) \mid \theta^{\Delta}\big) \equiv$ neural net with randomly initialized weights $\theta^{\Delta}$
         $\hat{\Delta} \equiv$ target network with $\theta^{\hat{\Delta}} \leftarrow \theta^{\Delta}$
 1 for all episode $= 1, N$ do
 2     for all step $= 1, M$ do
 3         observe $(s_t, a_t, s_{t+1})$
 4         store $(s_t, a_t, s_{t+1})$ in observation buffer $O$
 5     Sample random mini-batch of $\mathcal{T}$ observation pairs:
 6             $[(s_i, a_i, s_{i+1}), (s_j, a_j, s_{j+1})] \in O \times O$
 7     Set $y_{ij} \leftarrow (1 - \gamma)\delta^a(a_i, a_j) + \gamma\hat{\Delta}(s_{i+1}, s_{j+1} \mid \theta^{\hat{\Delta}})$
 8     Update $\Delta$ by minimizing the loss:
 9             $L = 1/\mathcal{T} \sum_{(i,j)} (y_{ij} - \Delta(s_i, s_j \mid \theta^{\Delta}))^2$
10     Every $C$ episodes update target network: $\theta^{\hat{\Delta}} \leftarrow \theta^{\Delta}$
```

8.2.3 Deep Behavioral Distance Learning Algorithm

Its aim is to fit parameters θ^{Δ} of a given neural net (we call it Δ-net) on observations that are sampled from a given simulation in a way that the trained Δ-net serves as a model for the distance function Δ. The target network can be seen as a twin of the Δ-net. It is holding snapshots of its sibling's parameter values which get synchronized every now and then (see Sect. 8.2.2). The intention is to stabilize the target values used in the recursive step within the update rule (as seen in Sect. 8.2.2; cf. the use of Δ_k for updating Δ_{k+1} in (8.3)) over a number of training episodes. It was shown that this approach helps to avoid divergence when iteratively updating the parameters of neural net weights [**mnih2015human-short**].

Training data is generated by observing the given simulation, and stored within the so called observation buffer (see Sect. 8.2.2). Note that an observed state s_t in fact comprises values for all the observable features within the simulation at a particular point in time t. As the observation buffer enables us to shuffle previously generated training data (in order to obtain random mini-batches as seen in Sect. 8.2.2), it helps us to break up unwanted temporal correlations that could have been introduced by the sequential manner of observation. On that basis, the Δ-net is trained to minimize the loss between its current and the wanted distance estimate (in terms of the update rule), averaged over mini-batches (Sect. 8.2.2). After learning, the Δ-net is supposed to return adequate distance estimates for arbitrary pairs of states that are included in the state space of the considered simulation.

8.3 Experimental Results

In order to evaluate the aforementioned ideas we considered a simulation of the described smart vacuum system (SVS) within a simplified environment consisting of a single rectangular room. Following the setting from previous experiments (cf. for instance Fig. 2.3) the simulated room was discretized to tiles of 1×1 meters and comprised the SVS itself, a rectangular stairwell as well as a mark the user is able to place at any tile with the intention that the SVS is driving towards. Though the room's and the stairwell's dimensions (width and height) as well as the position of stairwell ((x,y)-coordinates of the lower left tile) could be instantiated arbitrarily within the simulation, we assumed that all of these parameters would be fixed in reality. These thus formed the uncontrollable portion of our environmental state space. Consequently, test inputs consisted of the (x,y)-coordinates on which the mark was placed.

8.3.1 Setting

The mentioned environmental setting served as a framework for a first proof of concepts, but, of course, masks many details that would have to be considered when testing a real SVS. We thus did not reproduce any of the simulated states in reality. Instead, we used the simulation for both (1) defining test inputs and (2) applying these later on. As we motivated our distance metric by the claim that test inputs will usually have to be applied in real environments and that not all of the environmental features are controllable there, we restricted our simulation on fixed values of uncontrollable features in (2).

Within the simulation, we utilized, just like in the last chapter, reinforcement learning methods for learning the intended behavior of the SVS. We assumed the simplified action space of driving one tile north, south, west, or east. As we intended the system to drive directly towards the user-given mark not falling down the stairs, we rewarded the SVS by 1 for successfully reaching the user-given mark, but punished it for falling by -1.

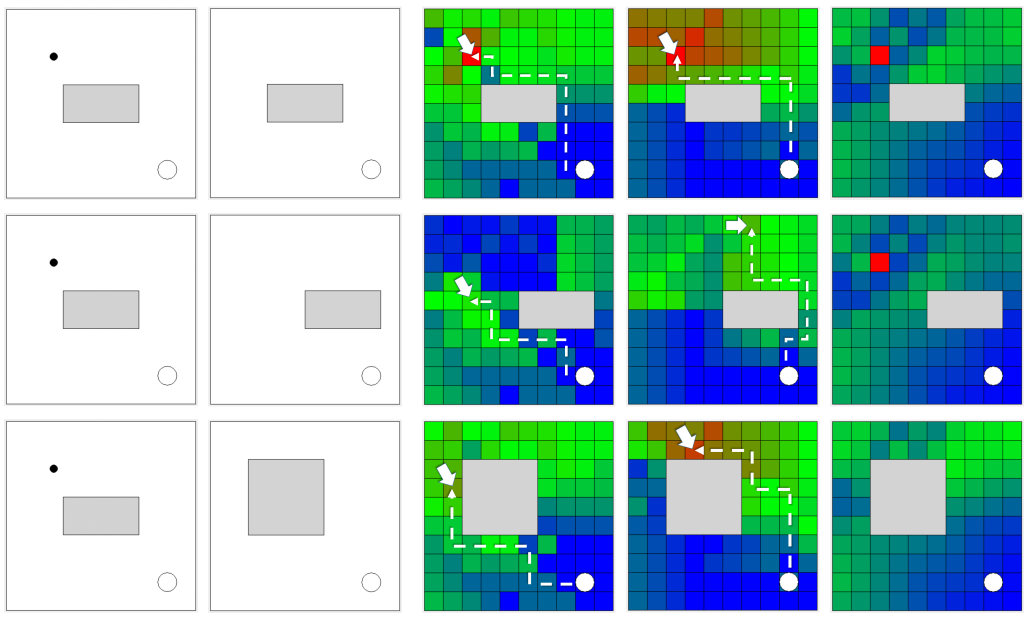
The resulting formalization of the SuT's intended behavior within the simulated environment was then used for executing some experiments. These were supposed to demonstrate that the presented distance function, indeed, helps to transfer test inputs between different environmental states in a reasonable way as well as that the DBDL algorithm is able to train a neural model for the distance function.

8.3.2 Reference Test Case

For the purpose of the experiments considered here, we just designed a reference test case by hand: We determined a room setup and placed the SVS as well as a mark on particular positions. The chosen setup is depicted in Fig. 8.1a. To emulate the challenge of dealing with uncontrollable features, as one would be confronted with when testing in reality, we afterward derived a set of similar but different environmental setups from the original test case.

Figure 8.1b shows three of these slightly adjusted setups, all of which are lacking the position of a mark. Based on the described experimental setup we confronted ourselves with the task of using the proposed distance function for determining mark positions within the modified environmental setups which, in combination with the given uncontrollable portion of the state, should result in preferably similar system behavior.

We used the following approach: First, we combined each of the modified, markless environmental setups with all potential mark positions. The distance function was then used to determine the distance between each of the resulting producible



(A) Original room setup comprising a mark

(B) Adjusted room setups

(C) Exhaustively calculated distance heat maps considering p_1

(D) Exhaustively calculated distance heat maps considering p_2

(E) Distance heat maps estimated by the Δ-net

(F) Distance colors from 0 (red) to 1 (blue)

FIGURE 8.1: Explicitly calculated heat maps for different room setups assuming a deterministic policy.

states and the original one. The state with the minimum distance was then supposed to include the requested mark position. To bypass estimation losses, we initially forewent the proposed learning algorithm, but calculated the distance of all the state-pairs exhaustively (following equation (8.2) with $\gamma = 0.8$). We thereby considered two different policies learned by separate reinforcement learning processes p_1 and p_2. Though both were trained on the same aforementioned reward function they show slightly different system behavior.

8.3.3 Results

Figures 8.1c und 8.1d show the obtained results. The *distance heat maps* that therein overlay the depicted room setups are meant for visualizing the calculated distance values: using the color range depicted in Fig. 8.1f, a tile's coloring represents the computed distance between the original (seen in Fig. 8.1a) and the modified state (seen in Fig. 8.1b) in which a mark was placed within the tile. The mark position which extended a modified setup (Fig. 8.1b) in the most similar way (w.r.t. the distance function) to the original one (Fig. 8.1a) is marked by the white arrow; the path that was taken by the SVS out of this state is sketched by the dotted white arrow.

As one can see, the modified state in the first line of Fig. 8.1b comprises the same setup as the original one. Thus, it seems reasonable that the same mark position for both policies would implement the most similar state (Figs. 8.1c und 8.1d). The heat map depicted in the second line of Fig. 8.1c is more surprising, since the position of the mark within the original state is considered as leading to dissimilar behavior within the modified version.

As the depiction, however, shows: the behavior, in fact, is dissimilar, as the robot now passes under the stairwell and could not follow the zigzag path it drove within the original setting afterward. By contrast, the policy obtained by p_2 proposes rather similar behavior for nearly all the possible mark positions as within the first setup modification — which leads to similar heat maps (cf. Fig. 8.1d). The heat maps shown in the bottom line of Figs. 8.1c und 8.1d resemble those in the top line again.

8.3.4 Evaluating the DBDL-algorithm

Based on these gained insights, we subsequently aimed at evaluating the proposed DBDL-algorithm. The idea was to compare the estimates of a neural network, which was trained by using the algorithm, to the exhaustively calculated distances mentioned above. To do so, a fully connected, feed forward neural net with two hidden layers (comprising 64 and 32 neurons) was trained as described in Alg. 5 (for comparability we used $\gamma = 0.8$ again). During the training process of 200 000 epochs, the observed simulation which just considered the system policy learned by reinforcement learning process p_1 was executed in randomly initialized environmental setups.

Figure 8.1e shows the distance heat maps which were estimated by the neural model after training. It appears that the neural model mostly generalized over the different modifications of the original state, as the heat maps are approximately equal. It seems that by learning to interpret the behavior of the SuT within arbitrary scenarios, with different dimensions and positions of the stairwell and different positions of the SVS itself, it learned to focus on the position of the mark.

This manifests itself particularly when comparing the results depicted in line 2 of Fig. 8.1c and Fig. 8.1e. As the zigzag course of the SuT seems only to occur in this particular setting, it does not affect the estimates of the neural net. Recalling that we explicitly gave a reward to the SVS for driving directly to the mark position, this can be seen as a kind of filtering outliers. This presumption is further supported by the impression that, although the net was exclusively trained on the policy obtained by p_1, the distance heat maps depicted in Fig. 8.1e resemble parts of both, those depicted in Fig. 8.1c as well as those depicted in Fig. 8.1d.

All in all, the experiments showed that the proposed distance metric can deliver on its promises: In cases where not all of the environmental features are controllable it seems promising to choose the test inputs that together with the uncontrollable portion of environmental features result in states that are preferably near to the intended one w.r.t. the defined function. As can be seen in Figs. 8.1c bis 8.1e, this will result in similar observable behavior of the SuT — at least considering the intended behavior. I believe that these evaluations are a solid basis for further investigations on how to fruitfully complement existing test suite derivation, coverage and, prioritization approaches by behavioral distance considerations.

8.4 Related Work

The SVS example described in the introduction points out the general challenge of transferring test inputs between simulated environments and reality. This challenge occurs when testing systems whose behavior at least partially depends on the environmental conditions. As we argued in [RK17b] that not all environmental features are controllable by the tester, we must assume that full equality on states between simulation and reality cannot be achieved. Common techniques of test design, including coverage metrics and test prioritization, are, however, mostly based on executable simulations or other kinds of models. Thus the question arises on how to retain the intention of the designed test inputs in reality.

Proposing a distance function on environmental states with different uncontrollable feature assignments, our answer to the aforementioned question was to produce the closest possible relatives of the designed test inputs in the partially controllable reality. Though the use of a distance function in principle is inspired by other domains, such as string comparisons (variants of the edit distance are common representatives there [RY98]), we provided a tester's view on the notion of distance which is rather domain-specific: the distance is determined by the intended behavior of the SuT.

From a machine learning point of view, the transfer task can be formulated as follows: To abstract the features of the state space in a way that the extracted, abstract features are describing the intention behind a produced state. Given such an abstraction, transferring state $s = \langle u, c \rangle$ from uncontrollable setup u to u' means to find an assignment of controllable features c' which together with u' is equal to c and u in terms of the abstract feature space.

In other words, we would like to learn a *generative* model for representing the joint probability $P(\langle u', c' \rangle, \phi)$ of states and equivalence classes in terms of equal abstract feature values ϕ [NJ01]. We need to do this in an *unsupervised* way, since training data is unlabeled (we assume not to have given examples of different state pairs representing an equal intention of the tester) [HTF09].

Generative Models In fact, there are common solutions for unsupervised learning of generative models, including *deep belief networks* [HOT06], *autoencoders* [HS06] and *generative adversarial networks* [Goo+14]. These, however, do not solve our particular issue, because of the following two characteristics:

1. We assume that not all of the environmental features are controllable. This non-controllability also implies limitations for the aforementioned machine learning process: suppose that we would like to transfer state $\langle u, c \rangle$ to another uncontrollable setup u', and that $\langle u, c \rangle$ is classified with the latent feature values ϕ. Because of the uncontrollable u' it could then be impossible to find a c' such that $\langle u', c' \rangle$ is also classified as ϕ.

   All of the mentioned generative solutions, however, assert that equality on the abstract features is always producible and thus cannot answer the question of what to do if not. A continuous distance metric, as we propose one, answers this question by returning the $\langle u', c' \rangle$ which minimizes the distance to $\langle u, c \rangle$.

2. We will usually not have enough training data to automatically extract reasonable abstract features (as it is done by all of the mentioned methods). Quite to the contrary, the limited number of test inputs we would like to transfer could permit various interpretations of their intention, probably most of which unreasonable. We overcame this problem by defining the intention of a test input heuristically but generically within the distance metric.

Inverse Reinforcement Learning Considering testing in general as a decision process over the action space of possible controllable feature assignments [RK16], the intention of choosing a particular assignment resembles the reward a tester is striving for. Using *reinforcement learning* techniques [SB98], an optimal policy could be learned by mapping states to the most promising assignment w.r.t. the reward. Within this reinforcement learning view, our task could be considered as a kind of *inverse reinforcement learning* [AN04], *imitation learning* [Sch99] or *transfer learning* [TS09] in general, as all these techniques are concerned with transferring knowledge between different decision processes.

However, the basis is different. While the mentioned approaches aim at simply copying actions of others (imitation learning), retracing reward functions of others (inverse reinforcement learning), or reusing internal knowledge on novel problems (transfer learning), we wanted to leverage knowledge about the behavior of others (the SuT) for translating particular, own actions (controllable features assignments) between environmental states without reconstructing the overall reward function.

Nevertheless, even if there is — to the best of my knowledge — no active work on this, I believe that there are several use cases in reinforcement learning domains for our issue, too: Whenever the own performance w.r.t. future reward depends on the behavior of others, as this is the case in games [Lit94] and multi-agent reinforcement learning tasks in general [BBDS08], the process of transferring own actions or policies (between different environments, tasks, etc.) should consider predictions of the others' reactions.

8.5 Conclusion

This chapter reported about experiments in which we tackled the challenge of transferring testing insights gained in simulated environments to the just partially controllable reality. For this purpose we suggested a behavioral distance metric. This compares pairs of test inputs based on the intended behavior of the SuT: The more different the SuT is supposed to respond to considered test inputs, the larger the distance of those. If behavioral considerations influenced test input derivation within a simulation (or another kind of test models), our measure can thus be used for finding the test inputs that are closest to the derived ones in reality — which is particularly useful when equality cannot be established there.

If the SuT's policy as well as the environmental transition probabilities are known the proposed distances can be obtained by iteratively refining (8.2) using the update rule depicted in (8.3). We however also covered the more realistic case where both are rather embedded within a simulation than known. In this case, the DBDL algorithm (Alg. 5) can learn to estimate the distance metric from observation.

First experiments which considered a simplified version of a SVS to be tested validated our claim: The distance function indeed triggered similar behavior of the SVS through different room setups. However, there are multiple opportunities and questions we cannot answer at this early point. In particular, the integration of the suggested distance function into existing testing approaches has to be investigated in the future, in order to evaluate its practical usefulness.

I envision two core areas of testing for this purpose: (1) test coverage and (2) test suite prioritization. For (1), I think about a kind of behavior-based coverage suggesting suites of test inputs that are as diverse as possible in terms of the behavioral distance (the notion of diversity for this scenario still has to be elaborated). If doing so, a test suite would trigger preferably different behavior variations of the SuT.

Using behavioral diversity as optimization measure in regression testing, such an approach could basically be used for test suite prioritization, too. Note that our technique transfers test inputs instead of rejecting them, as this is usual in current regression testing if environmental conditions have changed. In (2), I, however, rather think about complementing risk-based testing considerations by the distance function. This combination seems fruitful, as risk assessment highly depends on behavioral considerations which could be applied in reality by using our distance again.

I believe the aforementioned future tasks will also lead to some practical refinements for estimating the distance. While, so far, only a single instance of the one-step distance function δ was presented, one might think about several other variants which might include continuous action comparisons or risk considerations. Moreover, we did not consider optimizing the process of, given the distance function, determining the test input that is closest to another one until now. We still iterate over the set of all the candidates. This could be mitigated by using machine learning techniques again.

Chapter 9

Conclusion

With the collection of experiments considered in the last chapters this thesis should be seen as a first trace in a depth-first search for solutions in systematically testing self-adaptive systems. Following the premise that adaptive systems need adaptive test strategies a bunch of machine learning-based solutions were presented for different types of systems under test: we tested a *Self-Organizing Production Cell* using *Clustering* as well as *Evolutionary* algorithms; we utilized *Artificial Neural Networks* as continuous test models for a *Smart Energy Grid* and a *Smart Vacuum System* under test; *Reinforcement Learning Strategies*, and in particular the *Direct Future Prediction* approach was used for testing a smart vacuum system still learning at test time; and finally, we proposed the *Deep Behavior Distance Learning Algorithm* for adaptive test policy execution.

The considered systems under test should not be seen as a comprehensive enumeration of all types of self-adaptive systems and indeed it should be one part of future work to evaluate the presented approaches more deeply. Though, it is to state that there are still benchmarks for unified evaluations of testing approaches on self-adaptive systems missing. If there were similar evaluation frameworks for testing as there already are for training self-adaptive systems the GoT could get completed. If we consider the OpenAI Gym for instance, this inclusion of testing evaluation could be performed by providing an interface for manipulating and training the environment at run-time – just like it is already provided for training the respective learning systems.

Further branches of future work include particular improvements of methods presented before as well as more general discussions and maybe totally different solutions for the elaborated challenges. For the first, action items like a more detailed investigation of mutation operators for self-organizing, adaptive systems (see Sect. 4.6), the application of the learning approach from Sect. 5.3 on self-adaptive systems, imitating systems that are learning at run-time (see Sect. 6.5), addressing coverage-based test goals by DFP (see Sect. 7.5), and complementing risk assessments with metrics regarding the behavioral distance Sect. 8.5 had already been presented. Especially the application of DFP to diverse test goals will be investigated more deeply.

Further general directions of research for testing self-adaptive systems may include coverage metrics for systems learning at run-time (1), organizational aspects of quality assurance for self-adaptive systems (2) and the investigation of potential white-box approaches considering the knowledge base of self-adaptive systems (3).

Bibliography

[Agg+04] KK Aggarwal et al. "A neural net based approach to test oracle". In: *ACM SIGSOFT Software Engineering Notes* 29.3 (2004), pp. 1–6.

[Aho+91] Alfred V Aho et al. "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours". In: *IEEE Transactions on Communications* (1991), pp. 1604–1615.

[Aml00] Ståle Amland. "Risk-based testing: Risk analysis fundamentals and metrics for software testing including a financial application case study". In: *Journal of Systems and Software* 53.3 (2000), pp. 287–295.

[AN04] Pieter Abbeel and Andrew Y Ng. "Apprenticeship learning via inverse reinforcement learning". In: *Proceedings of the 21st International Conference on Machine Learning (ICML'04)*. ACM. 2004.

[AVMM95] Charles Anderson, Anneliese Von Mayrhauser, and Rick Mraz. "On the use of neural networks to guide software testing activities". In: *Proceedings of 1995 IEEE International Test Conference (ITC)*. IEEE. 1995, pp. 720–729.

[Bac99] James Bach. "Heuristic risk-based testing". In: *Software Testing and Quality Engineering Magazine* 11.9 (1999).

[Bau+08] Thomas Bauer et al. "Risikobasierte Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest". In: *Software Engineering* 121 (2008), pp. 99–111.

[BBDS08] Lucian Busoniu, Robert Babuska, and Bart De Schutter. "A comprehensive survey of multiagent reinforcement learning". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C* 38.2 (2008), pp. 156–172.

[Bel54] Richard Bellman. *The theory of dynamic programming*. Tech. rep. P-550. Rand Corp., 1954.

[Ben+14] Yoshua Bengio et al. "Deep Generative Stochastic Networks Trainable by Backprop". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proc. Machine Learning Research 2. PMLR, 2014, pp. 226–234.

[BJE94] Kirk Burroughs, Aridaman Jain, and Robert L Erickson. "Improved quality of protocol testing through techniques of experimental design". In: *Proceedings of ICC/SUPERCOMM'94 International Conference on Communications*. IEEE. 1994, pp. 745–752.

[Bla+05] Andreas Blass et al. “Play to test”. In: *Proceedings of the International Workshop on Formal Approaches to Software Testing*. Vol. 3997. Lect. Notes Comp. Sci. Springer, 2005, pp. 32–46.

[BMD09] Christos Boutsidis, Michael W Mahoney, and Petros Drineas. “An improved approximation algorithm for the column subset selection problem”. In: *Proceedings of the 20th annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics. 2009, pp. 968–977.

[BMK04] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. “Bi-criteria models for all-uses test suite reduction”. In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE. 2004, pp. 106–115.

[BN03] Bart Broekman and Edwin Notenboom. *Testing embedded software*. Pearson Education, 2003.

[BNB10] Laura Balzano, Robert Nowak, and Waheed Bajwa. “Column subset selection with missing data”. In: *NIPS workshop on low-rank methods for large-scale machine learning*. Vol. 1. Citeseer. 2010.

[Bou+07] Fabrice Bouquet et al. “A subset of precise UML for model-based testing”. In: *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM. 2007, pp. 95–104.

[BP13] Borja de Balle Pigem. “Learning Finite-State Machines — Statistical and Algorithmic Aspects”. PhD thesis. Universitat Politècnica de Catalunya, 2013.

[Bro+05] Manfred Broy et al. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Vol. 3472. Springer Science & Business Media, 2005.

[Bru+09] Yuriy Brun et al. “Engineering self-adaptive systems through feedback loops”. In: *Software engineering for self-adaptive systems*. Springer, 2009, pp. 48–70.

[Che+14] Betty H. C. Cheng et al. “Using models at runtime to address assurance for self-adaptive systems”. In: *Models@run. time (Dagstuhl Sem. 11481)*. Vol. 8378. Lect. Notes Comp. Sci. Springer, 2014, pp. 101–136.

[Che13] Nanxing Chen. “Passive Interoperability Testing for Communication Protocols”. PhD thesis. Université Rennes 1, 2013.

[Cho15] François Chollet. *Keras*. `https://github.com/fchollet/keras`. 2015.

[Cho16] François Chollet. *Building Autoencoders in Keras*. `https://blog.keras.io/building-autoencoders-in-keras.html`. 2016.

[Cho78] Tsun S. Chow. “Testing software design modeled by finite-state machines”. In: *IEEE transactions on software engineering* 3 (1978), pp. 178–187.

[CL96] Tsong Yueh Chen and Man Fai Lau. “Dividing strategies for the optimization of a test suite”. In: *Information Processing Letters* 60.3 (1996), pp. 135–141.

[Coh+03] Myra B Cohen et al. "Constructing test suites for interaction testing". In: *Proceedings of the 25th international conference on Software engineering*. IEEE Computer Society. 2003, pp. 38–48.

[Coh+97] David M. Cohen et al. "The AETG system: An approach to testing based on combinatorial design". In: *IEEE Transactions on Software Engineering* 23.7 (1997), pp. 437–444.

[Com+06] Autonomic Computing et al. "An architectural blueprint for autonomic computing". In: *IBM White Paper* 31 (2006), pp. 1–6.

[CS04] Christoph Csallner and Yannis Smaragdakis. "JCrasher: an automatic robustness tester for Java". In: *Software: Practice and Experience* 34.11 (2004), pp. 1025–1050.

[Dai04] Zhen Ru Dai. "Model-driven testing with UML 2.0". In: *Computer Science at Kent* (2004), p. 179.

[Dij72] Edsger W Dijkstra. "The humble programmer". In: *Communications of the ACM* 15.10 (1972), pp. 859–866.

[DK17] Alexey Dosovitskiy and Vladlen Koltun. "Learning to act by predicting the future". In: *Proceedings of the International Conference on Learning Representations*. 2017.

[DL+13] Rogério De Lemos et al. "Software engineering for self-adaptive systems: A second research roadmap". In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. Lect. Notes Comp. Sci. Springer, 2013, pp. 1–32.

[DLS78] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. "Hints on test data selection: Help for the practicing programmer". In: *Computer* 11.4 (1978), pp. 34–41.

[DN+07] Arilo C Dias Neto et al. "A survey on model-based testing approaches: a systematic review". In: *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM. 2007, pp. 31–36.

[DN84] Joe W Duran and Simeon C Ntafos. "An evaluation of random testing". In: *IEEE transactions on Software Engineering* 4 (1984), pp. 438–444.

[Doe16] Carl Doersch. "Tutorial on variational autoencoders". In: *arXiv preprint* (2016).

[Ebe+16] Benedikt Eberhardinger et al. "Back-to-back testing of self-organization mechanisms". In: *IFIP International Conference on Testing Software and Systems*. Springer. 2016, pp. 18–35.

[Ebe+17a] Benedikt Eberhardinger et al. "Adaptive tests for adaptive systems: The need for new concepts in testing for future software systems". In: *Softwaretechnik-Trends* 37.3 (2017).

[Ebe+17b] Benedikt Eberhardinger et al. "An approach for isolated testing of self-organization algorithms". In: *Software Engineering for Self-Adaptive Systems III. Assurances*. Springer, 2017, pp. 188–222.

[Ebe+17c] Benedikt Eberhardinger et al. "Test Case Selection Strategy for Self-Organization Mechanisms". In: *Test, Analyse und Verifikation von Software – gestern, heute, morgen*. dpunkt, 2017, pp. 139–157.

[EJ73] Jack Edmonds and Ellis L Johnson. "Matching, Euler tours and the Chinese postman". In: *Mathematical programming* 5.1 (1973), pp. 88–124.

[Ell86] Myrvin H Ellestad. *Stress Testing: Principles and Practice.* 1986.

[ES03] Agoston E Eiben and James E Smith. *Introduction to Evolutionary Computing*. Vol. 53. Natural Computing Series. Springer, 2003.

[ESS16] Ehsan Elhamifar, Guillermo Sapiro, and Shankar Sastry. "Dissimilarity-based sparse subset selection". In: *IEEE transactions on pattern analysis and machine intelligence* 38.11 (2016), pp. 2182–2197.

[ESV12] Ehsan Elhamifar, Guillermo Sapiro, and Rene Vidal. "Finding exemplars from pairwise dissimilarities via simultaneous sparse recovery". In: *Advances in Neural Information Processing Systems*. 2012, pp. 19–27.

[FA11] Gordon Fraser and Andrea Arcuri. "Evosuite: Automatic test suite generation for object-oriented software". In: *Proceedings of the 19th ACM SIGSOFT Symposium & 13th European Conference on Foundations of Software Engineering*. ACM. 2011, pp. 416–419.

[FD07] Brendan J Frey and Delbert Dueck. "Clustering by passing messages between data points". In: *science* 315.5814 (2007), pp. 972–976.

[FDC14] Erik M Fredericks, Byron DeVries, and Betty H. C. Cheng. "Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty". In: *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM. 2014, pp. 17–26.

[FHR13] Ylies Falcone, Klaus Havelund, and Giles Reger. "A Tutorial on Runtime Verification." In: *Engineering Dependable Software Systems* 34 (2013), pp. 141–175.

[FM00] Justin E Forrester and Barton P Miller. "An empirical study of the robustness of Windows NT applications using random testing". In: *Proceedings of the 4th USENIX Windows System Symposium*. Vol. 4. Seattle. 2000, pp. 59–68.

[FRC13] Erik M Fredericks, Andres J Ramirez, and Betty H. C. Cheng. "Towards run-time testing of dynamic adaptive systems". In: *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'13)*. IEEE. 2013, pp. 169–174.

[FS12] Eugene A Feinberg and Adam Shwartz. *Handbook of Markov Decision Processes: Methods and Applications*. Vol. 40. Springer Science & Business Media, 2012.

[Fuj+91] Susumu Fujiwara et al. "Test selection based on finite state models". In: *IEEE Transactions on software engineering* 17.6 (1991), pp. 591–603.

[FW07] Gordon Fraser and Franz Wotawa. "Redundancy based test-suite reduction". In: *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*. Vol. 4422. Lect. Notes Comp. Sci. Springer, 2007, pp. 291–305.

[Gal04] Daniel Galin. *Software quality assurance: from theory to implementation*. Pearson Education India, 2004.

[Gam+15] Erich Gamma et al. *Design Patterns: Elements Of Reusable Object-Oriented Software*. Pearson India, 2015.

[Gar05] Angelo Gargantini. "Conformance Testing". In: *Model-based testing of reactive systems*. Ed. by Manfred Broy et al. Springer Science & Business Media, 2005.

[GBLP18] Thomas Gabor, Lenz Belzner, and Claudia Linnhoff-Popien. "Inheritance-Based Diversity Measures for Explicit Convergence Control in Evolutionary Algorithms". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2018.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F Andler. "Combination testing strategies: a survey". In: *Software Testing, Verification and Reliability* 15.3 (2005), pp. 167–199.

[Goo+14] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680.

[Gre92] John J Grefenstette. "Genetic algorithms for changing environments". In: *Proceedings of Parallel Problem Solving from Nature 2*. Elsevier, 1992, pp. 139–146.

[Gro11] Alex Groce. "Coverage rewarded: Test input generation via adaptation-based programming". In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2011, pp. 380–383.

[Güd+08] Matthias Güdemann et al. "A specification and construction paradigm for organic computing systems". In: *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE. 2008, pp. 233–242.

[Hab+15] Axel Habermaier et al. "Runtime Model-Based Safety Analysis of Self-Organizing Systems with S#". In: *IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. IEEE. 2015, pp. 128–133.

[Ham50] Richard W Hamming. "Error detecting and error correcting codes". In: *Bell Labs Technical Journal* 29.2 (1950), pp. 147–160.

[Hau+10] Matthew Hause et al. "Testing safety critical systems with SysML/UML". In: *15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE. 2010, pp. 325–330.

[Hay09] Simon Haykin. *Neural networks and learning machines/Simon Haykin.* New York: Prentice Hall, 2009.

[Hei66] Robert A Heinlein. *The moon is a harsh mistress*. Macmillan, 1966.

[HLR15] Axel Habermaier, Johannes Leupolz, and Wolfgang Reif. "Executable Specifications of Safety-Critical Systems with S#". In: *IFAC-PapersOnLine* 48.7 (2015), pp. 44–49.

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural Computation* 18.7 (2006), pp. 1527–1554.

[HP85] David Harel and Amir Pnueli. "On the development of reactive systems". In: *Logics and Models of Concurrent Systems*. Ed. by Krzysztof R. Apt. Springer, 1985, pp. 477–498.

[HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *Science* 313.5786 (2006), pp. 504–507.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. "Unsupervised learning". In: *The elements of statistical learning*. Springer, 2009, pp. 485–585.

[HVG15] Joachim Hänsel, Thomas Vogel, and Holger Giese. "A Testing Scheme for Self-Adaptive Software Systems with Architectural Runtime Models". In: *IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*. IEEE. 2015, pp. 134–139.

[IS03] Hisao Ishibuchi and Youhei Shibata. "A similarity-based mating scheme for evolutionary multiobjective optimization". In: *Genetic and Evolutionary Computation Conference*. Vol. 2723. Springer, 2003, pp. 1065–1076.

[Jen+18] Rikke Hagensby Jensen et al. "Designing the desirable smart home: A study of household experiences and energy consumption impacts". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM. 2018, p. 4.

[JH03] James A Jones and Mary Jean Harrold. "Test-suite reduction and prioritization for modified condition/decision coverage". In: *IEEE Transactions on Software Engineering* 29.3 (2003), pp. 195–209.

[JH08] Yue Jia and Mark Harman. "Constructing subtle faults using higher order mutation testing". In: *8th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2008, pp. 249–258.

[JRR15] Marcin Jamro, Dariusz Rzonca, and Wojciech Rząsa. "Testing communication tasks in distributed control systems with SysML and Timed Colored Petri Nets model". In: *Computers in Industry* 71 (2015), pp. 77–87.

[Kan+07] S Kanmani et al. "Object-oriented software fault prediction using neural networks". In: *Information and software technology* 49.5 (2007), pp. 483–492.

[KHE11] Johannes Kloos, Tanvir Hussain, and Robert Eschbach. "Risk-based Testing of Safety-Critical Embedded Systems Driven by Fault Tree Analysis". In: *IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2011, pp. 26–33.

[KK+12] Mohd Ehmer Khan, Farmeena Khan, et al. "A comparative study of white box, black box and grey box testing techniques". In: *International Journal of Advanced Computer Science and Applications* 3.6 (2012).

[KL16] Anne Kramer and Bruno Legeard. *Model-Based Testing Essentials-Guide to the ISTQB Certified Model-Based Tester: Foundation Level*. John Wiley & Sons, 2016.

[Klu+16] Dominik Klumpp et al. "Optimising Runtime Safety Analysis Efficiency for Self-Organising Systems". In: *IEEE International Workshops on Foundations and Applications of Self* Systems*. IEEE. 2016, pp. 120–125.

[KPM92] Taghi M Khoshgoftaar, Abhijit S Pandya, and Hemant B More. "A neural network approach for predicting software development faults". In: *Proceedings of the 3rd International Symposium on Software Reliability Engineering*. IEEE. 1992, pp. 83–89.

[KPV03] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. "Generalized symbolic execution for model checking and testing". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2003, pp. 553–568.

[KR87] Leonard Kaufman and Peter Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.

[Kul+15] Tejas D Kulkarni et al. "Deep convolutional inverse graphics network". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2539–2547.

[KW14] Diederik P Kingma and Max Welling. "Stochastic gradient VB and the variational auto-encoder". In: *2nd International Conference on Learning Representations (ICLR)*. 2014.

[LBP94] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. "Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method". In: *IEEE Transactions on Software Engineering* 2 (1994), pp. 149–162.

[Lem+13] Rogério de Lemos et al. "Software engineering for self-adaptive systems: A second research roadmap". In: *Software Engineering for Self-Adaptive Systems II*. Vol. 7475. Lect. Notes Comp. Sci. Springer, 2013, pp. 1–32.

[Lit94] Michael L Littman. "Markov games as a framework for multi-agent reinforcement learning". In: *Proceedings of the 11th International Conference on Machine Learning (ICML 1994)*. 1994, pp. 157–163.

[LM05] Daniel Lowd and Christopher Meek. "Adversarial learning". In: *Proceedings of the 11th ACM SIGKDD Intlernational Conference on Knowledge Discovery in Data Mining*. ACM. 2005, pp. 641–647.

[LP97] Harry R Lewis and Christos H Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, 1997.

[LR02] Marco Locatelli and Ulrich Raber. "Packing equal circles in a square: a deterministic global optimization approach". In: *Discrete Applied Mathematics* 122.1 (2002), pp. 139–166.

[LS02] Michael L Littman and Richard S Sutton. "Predictive representations of state". In: *Advances in Neural Information Processing Systems*. 2002, pp. 1555–1561.

[LY96] David Lee and Mihalis Yannakakis. "Principles and methods of testing finite state machines-a survey". In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123.

[Mao+06] Ye Mao et al. "Neural networks based automated test oracle for software testing". In: *International Conference on Neural Information Processing*. Springer. 2006, pp. 498–507.

[McM04] Phil McMinn. "Search-based software test data generation: a survey". In: *Software Testing, Verification and Reliability* 14.2 (2004), pp. 105–156.

[ME+13] Frank D Macías-Escrivá et al. "Self-adaptive systems: A survey of current approaches, research challenges and applications". In: *Expert Systems with Applications* 40.18 (2013), pp. 7267–7279.

[Mea55] George H Mealy. "A method for synthesizing sequential circuits". In: *Bell Labs Technical Journal* 34.5 (1955), pp. 1045–1079.

[Mey88] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice Hall, 1988.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2011.

[Nac+04] Lev Nachmanson et al. "Optimal strategies for testing nondeterministic systems". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 4. ACM. 2004, pp. 55–64.

[Naf12] Florian Nafz. "Verhaltensgarantien in selbst-organisierenden Systemen". PhD thesis. Universität Augsburg, 2012.

[Ngu+09] Cu D Nguyen et al. "Evolutionary testing of autonomous software agents". In: *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (Vol. 1)*. Intl. Found. Autonomous Agents and Multiagent Systems. 2009, pp. 521–528.

[Ngu+12] Cu D Nguyen et al. "Evolutionary testing of autonomous software agents". In: *Autonomous Agents and Multi-Agent Systems* 25.2 (2012), pp. 260–283.

[NJ01] Andrew Y. Ng and Michael I. Jordan. "On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes". In: *Proceedings of the 14th Annual Conference on Advances in Neural Information Processing Systems (NIPS'01)*. MIT, 2001, pp. 841–848.

[OA99] Jeff Offutt and Aynur Abdurazik. "Generating tests from UML specifications". In: *International Conference on the Unified Modeling Language*. Springer. 1999, pp. 416–429.

[OB88] Thomas J. Ostrand and Marc J. Balcer. "The category-partition method for specifying and generating fuctional tests". In: *Communications of the ACM* 31.6 (1988), pp. 676–686.

[Off89] A. Jefferson Offutt. "The coupling effect: fact or fiction". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 14. 8. ACM. 1989, pp. 131–140.

[OH96] A Jefferson Offutt and J Huffman Hayes. "A semantic model of program faults". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 21. 3. ACM. 1996, pp. 195–200.

[OPV95] Jeff Offutt, Jie Pan, and Jeffrey M Voas. "Procedures for reducing the size of coverage-based test sets". In: *Proceedings of the 12th International Conference on Testing Computer Software*. ACM Press New York. 1995, pp. 111–123.

[Pin+17] Lerrel Pinto et al. "Robust adversarial reinforcement learning". In: *arXiv preprint arXiv:1703.02702* (2017).

[Pla16] Matthias Plappert. *keras-rl*. `https://github.com/keras-rl/keras-rl`. 2016.

[PM04] Bruce Potter and Gary McGraw. "Software security testing". In: *IEEE Security & Privacy* 2.5 (2004), pp. 81–85.

[Pre05] Alexander Pretschner. "Model-based testing". In: *Proceedings of the 27th International Conference on Software Engineering*. IEEE. 2005, pp. 722–723.

[PS98] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[PYB96] A Petrenko, N Yevtushenko, and G v Bochmann. "Testing deterministic implementations from nondeterministic FSM specifications". In: *Testing of Communicating Systems*. Springer, 1996, pp. 125–140.

[Rei+16] André Reichstaller et al. "Risk-based interoperability testing using reinforcement learning". In: *IFIP International Conference on Testing Software and Systems*. Springer. 2016, pp. 52–69.

[Rei+17] André Reichstaller et al. "Applying deep learning for imitating adaptive agent behavior in statistical software testing". In: (2017).

[Rei+18] André Reichstaller et al. "Test Suite Reduction for Self-organizing Systems: A Mutation-based Approach". In: *IEEE/ACM 13th International Workshop on Automation of Software Test* (2018).

[RGK18] André Reichstaller, Thomas Gabor, and Alexander Knapp. "Mutation-based test suite evolution for self-organizing systems". In: *International Symposium on Leveraging Applications of Formal Methods*. Springer. 2018, pp. 118–136.

[Rie05] Martin Riedmiller. "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method". In: *Proceedings of the 16th European Conference on Machine Learning (ECML'05)*. Vol. 3720. Lect. Notes Comp. Sci. Springer. 2005, pp. 317–328.

[RK16] André Reichstaller and Alexander Knapp. "Hunting the Game: Towards a game of testing adaptive systems". In: *International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2016, pp. 303–304.

[RK17a] André Reichstaller and Alexander Knapp. "Compressing Uniform Test Suites Using Variational Autoencoders". In: *International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE. 2017, pp. 435–440.

[RK17b] André Reichstaller and Alexander Knapp. "Transferring Context-Dependent Test Inputs". In: *International Conference on Software Quality, Reliability and Security*. IEEE. 2017, pp. 65–72.

[RK18] André Reichstaller and Alexander Knapp. "Risk-based Testing of Self-Adaptive Systems using Run-Time Predictions". In: *IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE. 2018, pp. 80–89.

[Rot+01] Gregg Rothermel et al. "Prioritizing test cases for regression testing". In: *IEEE Transactions on software engineering* 27.10 (2001), pp. 929–948.

[Rot+02] Gregg Rothermel et al. "Empirical studies of test-suite reduction". In: *Software Testing, Verification and Reliability* 12.4 (2002), pp. 219–249.

[RY98] Eric Sven Ristad and Peter N Yianilos. "Learning string-edit distance". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.5 (1998), pp. 522–532.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[Sch99] Stefan Schaal. "Is imitation learning the route to humanoid robots?" In: *Trends in Cognitive Sciences* 3.6 (1999), pp. 233–242.

[SEFY07] Natalia Shabaldina, Khaled El-Fakih, and Nina Yevtushenko. "Testing nondeterministic finite state machines with respect to the separability relation". In: *Testing of Software and Communicating Systems*. Springer, 2007, pp. 305–318.

[Sha+11] Seyed Reza Shahamiri et al. "An automated framework for software test oracle". In: *Information and Software Technology* 53.7 (2011), pp. 774–788.

[Shi+14] August Shi et al. "Balancing trade-offs in test-suite reduction". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 246–256.

[Sin+03] Satinder P Singh et al. "Learning predictive state representations". In: *Proceedings of the 20th International Conference on Machine Learning*. 2003, pp. 712–719.

[SKW+15] Tim Salimans, Diederik P Kingma, Max Welling, et al. "Markov Chain, Monte Carlo, and Variational Inference: Bridging the Gap". In: *International Conference on Machine Learning (ICML)*. 2015, pp. 1218–1226.

[SLS14] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, Inc., 2014.

[SM07] Heiko Stallbaum and Andreas Metzger. "Employing Requirements Metrics for Automating Early Risk Assessment". In: *Workshop on Measuring Requirements for Project and Product Success (MeReP)*. 2007, pp. 1–12.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM. 2005, pp. 263–272.

[Spi+17] Helge Spieker et al. "Reinforcement learning for automatic test case prioritization and selection in continuous integration". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM. 2017, pp. 12–22.

[Ste+13] Jan-Philipp Steghöfer et al. "A System of Systems Approach to the Evolutionary Transformation of Power Management Systems". In: *Informatik 2013 – Workshop "Smart Grids"*. Lecture Notes in Informatics. Bonner Köllen Verlag, 2013.

[Tre08] Jan Tretmans. "Model based testing with labelled transition systems". In: *Formal methods and testing*. Springer, 2008, pp. 1–38.

[Tre92] G.J. Tretmans. *A Formal Approach to Conformance Testing*. 1992.

[Tro09] Joel A Tropp. "Column subset selection, matrix factorization, and eigenvalue optimization". In: *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics. 2009, pp. 978–986.

[TS09] Matthew E. Taylor and Peter Stone. "Transfer learning for reinforcement learning domains: A survey". In: *Journal of Machine Learning Research* 10 (2009), pp. 1633–1685.

[UMN97] Iwan Ulrich, Francesco Mondada, and J.-D. Nicoud. "Autonomous vacuum cleaner". In: *Robotics and Autonomous Systems* 19.3-4 (1997), pp. 233–245.

[UPL12] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (2012), pp. 297–312.

[Urs02] Rasmus K Ursem. "Diversity-guided evolutionary algorithms". In: *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*. 2002, pp. 462–471.

[Vin+08] Pascal Vincent et al. "Extracting and composing robust features with denoising autoencoders". In: *International Conference on Machine Learning (ICML)*. ACM. 2008, pp. 1096–1103.

[VLK02] Meenakshi Vanmali, Mark Last, and Abraham Kandel. "Using a neural network in the software testing process". In: *International Journal of Intelligent Systems* 17.1 (2002), pp. 45–62.

[VRC06] Margus Veanes, Pritam Roy, and Colin Campbell. "Online testing with reinforcement learning". In: *Formal Approaches to Software Testing and Runtime Verification*. Springer, 2006, pp. 240–253.

[Wal+16] Jacob Walker et al. "An uncertain future: Forecasting from static images using variational autoencoders". In: *European Conference on Computer Vision (ECCV)*. Springer. 2016, pp. 835–851.

[Was+18] Bernd Waschneck et al. "Optimization of global production scheduling with deep reinforcement learning". In: *Procedia CIRP* 72.1 (2018), pp. 1264–1269.

[Wat89] Christopher John Cornish Hellaby Watkins. "Learning from Delayed Rewards". PhD thesis. King's College, Cambridge, 1989.

[WKS12] M. F. Wendland, M. Kranz, and Ina Schieferdecker. "A Systematic Approach to Risk-based Testing Using Risk-annotated Requirements Models". In: *7th International Conference on Software Engineering Advances (ICSEA)*. 2012, pp. 636–642.

[WL05] Stefan Wappler and Frank Lammermann. "Using evolutionary algorithms for the unit testing of object-oriented software". In: *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. ACM. 2005, pp. 1053–1060.

[WP01] Alan W Williams and Robert L Probert. "A measure for component interaction test coverage". In: *ACS/IEEE International Conference on Computer Systems and Applications*. IEEE. 2001, pp. 304–311.

[WT94] James A Whittaker and Michael G Thomason. "A Markov chain model for statistical software testing". In: *IEEE Transactions on Software Engineering* 20.10 (1994), pp. 812–824.

[WV00] Elaine J Weyuker and Filippos I Vokolos. "Experience with performance testing of software systems: issues, an approach, and case study". In: *IEEE transactions on software engineering* 26.12 (2000), pp. 1147–1156.

[Yan00] Mihalis Yannakakis. "Hierarchical state machines". In: *IFIP International Conference on Theoretical Computer Science*. Springer. 2000, pp. 315–330.

[Yan04] Mihalis Yannakakis. "Testing, optimization, and games". In: *Proceedings of the 19th Annual Symposion on Logic in Computer Science (LICS 2004)*. IEEE. 2004, pp. 78–88.

[Yaq+19] Ibrar Yaqoob et al. "Autonomous Driving Cars in Smart Cities: Recent Advances, Requirements, and Challenges". In: *IEEE Network* (2019).

[YH12] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: a survey". In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

[Zha+11] Lingming Zhang et al. "An empirical study of JUnit test-suite reduction". In: *Proceedings of the 22nd International Symposium on Software Reliability Engineering*. IEEE. 2011, pp. 170–179.

[Zim+09] Fabian Zimmermann et al. "Risk-based Statistical Testing: A Refinement-based Approach to the Reliability Analysis of Safety-Critical Systems". In: *12th European Workshop on Dependable Computing (EWDC)*. 2009.