# Testing self-adaptive systems - an overview

Tilo Werdin
Dominik Olwig

## Contents



## 1. Abstract

During the past years, the requirements for software-systems have changed dramatically. Software now often needs to be self-adaptive. This also leads to a different process of development. One important aspect of developing software is testing. Due to the unflexibility of traditional tests, there is a need for new testing-methods. There are some approches, that target this problem. What is missing is an overview about them.
Therefore we want to categorize, shortly describe and evaluate the different ideas, that exist in current literature.
In order to find as many different approches as possible, we apply the snow-balling-method during our research and we structure them using different categorization techniques.
The outcome of our research will be a taxonomy and an evaluation of state of the art testing techniques for self-adaptive systems.

Keywords: self-adaptive systems; system testing

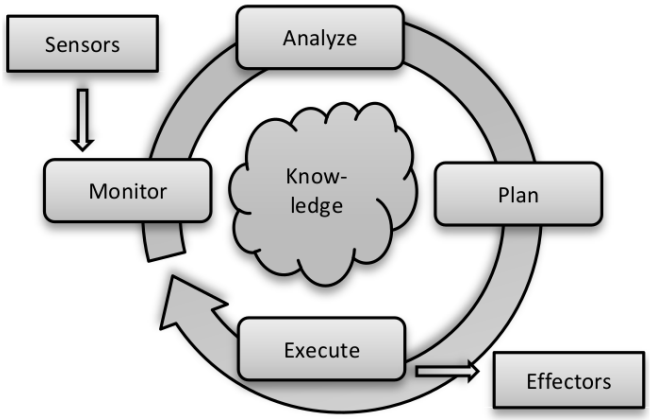


**Figure 1.** MAPE [2]

## 2. Introduction

Digital systems are more and more integrated in our daily lives. This makes high demands for software-systems, because the software needs to be able to adapt to different environments. A system that is able to do this is called a self-adaptive system (SAS).

There are multiple applications for SAS in different domains such as digital assistants, self-driving cars, highly distributed web-services or robots to name a few. Some of them are safety-critical applications, that could cause high damage and even harm people, if they behaved wrongly. For this reason it is very important to test these systems to ensure good behavior.

But before thinking about testing, it is important to understand how SAS work and what characterizes them. The main idea of SAS is the control loop. A control loop contains four main parts (as shown in Figure 1): monitoring, analysis, planning and execution (MAPE). The control loop operates frequently and influences the behavior of the system. While monitoring and execution interact directly with physical parts of the system, the analysis and planning parts of the loop calculate the adjustments the system should make.

A SAS is often characterised by a high degree of distribution. A system can use third party services and is able to change between multiple services at runtime. These external services were developed and deployed by different stakeholders.

These properties of self-adaptive systems lead to many challanges regarding testing of these systems. One consequence is that system-testers do not have total access to the code the system will execute. Moreover a tester can not predict at a specific call-site, which code will be executed next, because that depends on the currently used service and environment. The complexity is an other big challange for testing SAS. Every different environment and state of the system can lead to different desired behavior, which causes a combinatorical explosion of test-cases. Traditional testing assumes, that a system will always behave in a good way, when all possible test-cases were tested successfully. But this approach is not practicable for SAS. The amount of test-cases is too high and it might be impossible to execute some tests, because the tester does not have all code parts that will be executed.

To find existing testing-methods that target these problems, we used the snowballing-method during our research. We then grouped the papers we had found regarding their main approach.

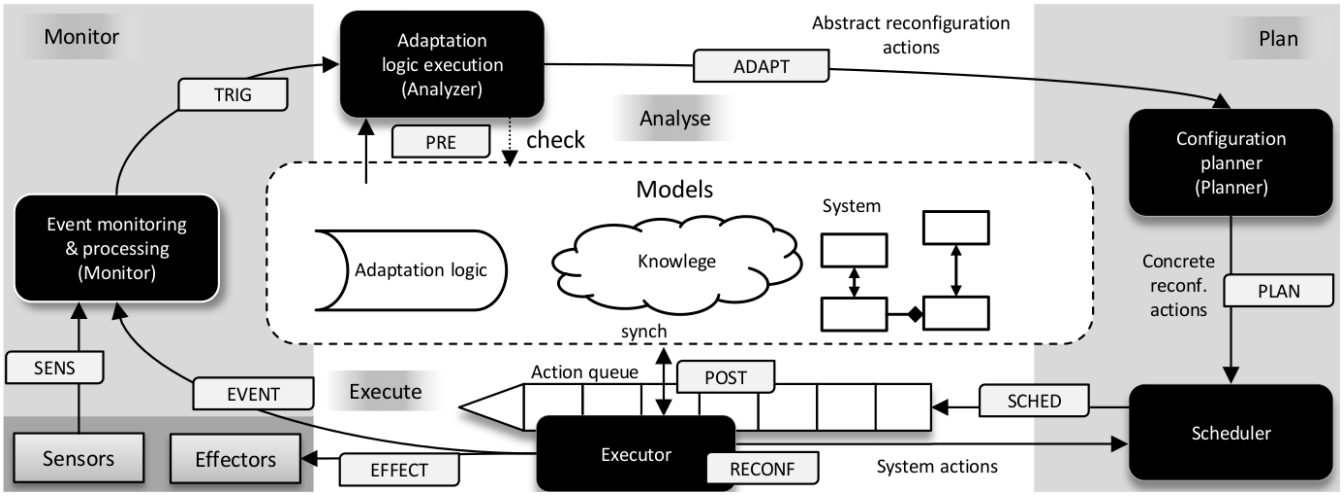


**Figure 2.** occurence of failures

## 3. Two approaches for testing self-adaptive systems

### 3.1 Classification overview

All testing methods that we found during our research can be classified into two main ideas: self-testing and modelbased testing. The following sections will shortly explain the different methods.

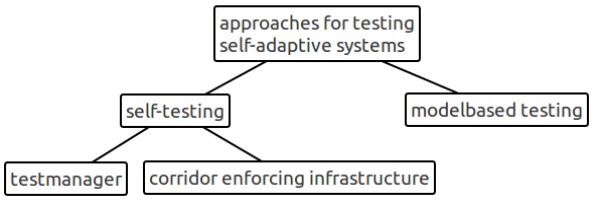


**Figure 3.** classification of test approaches

### 3.2 Self-Testing

Self-testing handles the adaptive character of SAS by monotoring the system at runtime and checking for violations of constraints. These constraints can be results of the system or quality of service constraints such as: response time. When bad behavior is observed by the test-environment, it tries to execute a recovery-action to bring the system back into a good state that produces the desired results.
During our research we found two approaches that can be classified into self-testing: testmanager and corridor enforcing infrastructure.

#### 3.2.1 Testmanager

One interesting idea for a self-test architecture is a testmanager. In general this is a software-component, that runs simultanously to the monitored system. Actions that regard to adaption of the system need to be tested by this manager during runtime. The adaption will just be committed, if all tests were passed successfully.

There are two variants of using a testmanager:

1. safe adaption with validation
Every time the system perceives a contextual change it notifies an internal adaptation-manager. It decides wheather an adaption is needed. If the system wants to adapt, the adaptation-manager will initiate an adaption and at the same time notify the testmanager. After the adaptation is completed, all actions targeting the system are blocked. In the meanwhile the testmanager is executing a set of tests that depend on requirements that the adaption-manager sent. When all tests were finished the result is sent back to the system. The systems adaptation-manager will keep the changes, if the tests were successful or it will recover the old system-state, if a test failed.

2. replication with validation
The main idea of this architecture is similar to the idea of safe adaption with validation. When the system needs to do an adaptation it notifies a test-manager and it executes an adaption. But in contrast to the idea of “safe adaption with validation“ the adaption is not executed directly on the running service. Insead, a copy of the service is created and the adaptation is executed on the copy. The tests are then performed on the copy. At the same time, incoming requests are handled by the old system. After all tests finished successfully, the copy gets the new active handler of requests. If the test fails, the copy can be dropped.

#### 3.2.2 Corridor Enforcing Infrastructure

An other self-testing approach, that focuses more on continuous monitoring of system constraints, is the corridor enforcing infrastructure. A system behaves in a good way, if it fulfills all its constraints at any time. As an illustration one

could call this range of good system states defined by these constraints a “Corridor“. Eberhardinger et al. [1] have introduced an infrastructure, that continually monitores the system state and checks wheather the current state is still within the corridor of correct behavior (CCB). If the systems leaves the CCB, the test-system would need to bring the system back to a good state.

### 3.3 Model-based testing

Modelbased testing makes traditional unit tests applicable for self-adaptive systems by generating tests automatically. The idea of model-based testing is not to think of all different scenarios and writing tests to each test-case oneself, but to generate these tests based on a model of the system. The target of generating tests for SAS is to cover all available scenarios and to test every situation the SAS might be in.

#### 3.3.1 Finding failure scenarios

To find failure scenarios there is the need for searching for aspects that can fail during execution. Püschel et al. [2] define them refering to Figure 2. The different failure scenarios are:

- SENS: misinterpreted sensor data
- TRIG: misinterpreted event
- PRE: misinterpreted model
- ADAPT: wrong adaptation derived
- PLAN: inconsistent planning
- SCHED: inconsistent scheduler
- POST: corrupt model construction
- RECONF: reconfiguration failure
- EVENT: wrong event creation
- EFFECT: processing wrong effect production

#### 3.3.2 Modelling of behavior

The next step is to model the behavior of the SAS. Therefore it is useful to look at all the failure scenarios and derive some behavior models from that. Models could for example be some state-flow charts that show which state should follow after a certain state. Especially interesting now are state-flows that seem to be orthogonal.

An example of how this could look like was made by Püschel et al. in [3].

#### 3.3.3 Testing

After modelling different orthogonal state-flows or other behavior models, the test-suite will do the rest of the work: generating test-cases. This is done by merging the models together canonically and getting a huge set of test scenarios. For each of these test scenarios there will be one separate test. The great benefit is that the test engineer does not need to think of every scenario. When the behavior is modeled, the test-suite will generate every single test scenario that can be derived from the models. So if there is a test-case that is very unlikely to happen, the testing engineer might leave it out or does not even think about it. But with this method it is tested anyway. If the system then gets to this unlikely state it will be prepared because the scenario was handled during testing already.

# 4. Evaluation

## 4.1 Criteria

For our evaluation of the different testing strategies, we need some meaningful criteria. With the help of these criteria we can rate and compare the strategies. We give every criteria for every approach a rating between 1 and 5, where 5 is the best possible behavior, that one can think of for the concrete criteria.
Below is the list of our criteria with explanations:

- control during runtime... amount of impact of the test-suite during execution
- ensure quality before deploy... how much quality can the test-suite ensure before the system gets deployed
- performance overhead... amount of additional effort for running the test-suite simultanously to the system, which includes time and memory
- testing-cost... how complex is building the test for the developer
- adaptability... how easy can the test-suite be adapted to an other system

## 4.2 Comparison

### 4.2.1 Testmanager

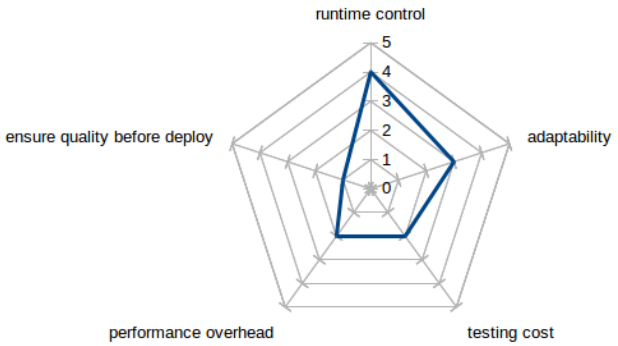


**Figure 4.** testmanager - Kiviat graph

At first we want to evaluate the testmanager approach. A testmanager has the ability to handle adaptation of the system and therefore its runtime control is high. It has not the best possible score, because there are situation, where the manager can not handle a test violoation (e.g. injured response time constraint). On the other hand it can not ensure good behavior prior to release, because there are no tests before deployment. All tests were performed at runtime, which causes an overhead. By using the “safe adaption with validation“ method there is a higher time overhead and by using the “replication with validation“ methode there is a bigger memory overhead. Due to its independance from the monitored system, the testmanager is relatively flexible and can be used as an independent component. A system developer would need to define the constraints for each situation and the testmanager would automatically perform the required tests at runtime. Defining these constraints can be very time-consuming, if there are multiple different environments and actors that need to be included in the testing process.

### 4.2.2 Corridor enforcing infrastructure

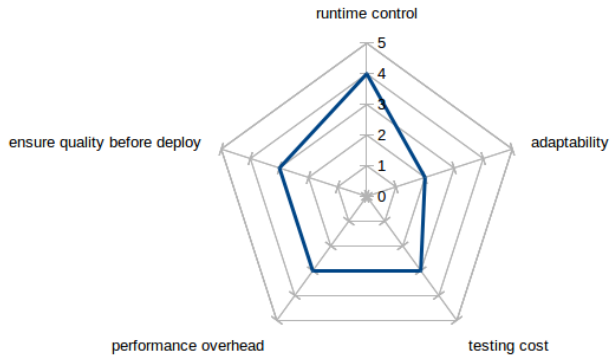


**Figure 5.** corridor enforcing infrastructure - Kiviat graph

Next we will evaluate the corridor enforcing infrastructure (CEI). Like the testmanager, it is also able to adapt to a new system-state and can execute well fitting tests. As well as the testmanager it has not the highest possible score, because the CEI might not always be able to recover a good system-state. The CEI highly depends on the system, because the CEI must know possible actions of the system to bring it back to a save state. This reduces the adaptability and makes the testing-costs very high. As a result of the big infrastructure that need to run, there is a performance overhead. One advantage of the CEI over the testmanager is, that it can ensure quality before deploy to some degree. The reason is that the CEI itself can be tested before deploy. If one can prove that the CEI is working it will hold the system within the corridor of correct behavior. But testing the CEI is difficult, because one has to simulate system behavior and check the CEIs reactions.

### 4.2.3 modelbased testing

Modelbased testing tries to ensure good behavior by testing prior to deployment. That is why modelbased testing can ensure quality before deploy. Due to the fact, that no additional testing process needs to run during execution of the system, there is no performance overhead for the running SAS. On the other hand this leads to low runtime control, because the test-suite can not react to wrong behavior. An existing model for a system can partly be applied to new similar systems. The more similar both systems are the better the model can be applied. The time consuming writing of test cases is executed by a generator which causes relatively low testing costs.

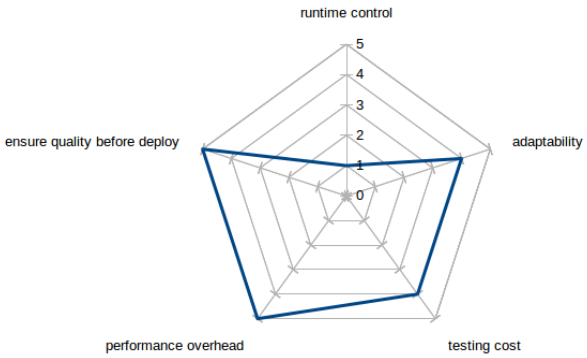


**Figure 6.** model based testing - Kiviat graph

## 5. Future Work

Our research has shown: Testing self-adaptive systems is very difficult but good methods were needed for present-day systems. Currently there are two main-approaches: self-testing and modelbased testing. Each of them has its own advantages and disadvantages. Self-testing is good at handling adaptions and controling the system at runtime. On the other hand, it is difficult to ensure good behavior before deploy. Modelbased testing is good at asserting good system behavior prior to release, but can not control the system at runtime.

A combination of both methods could lead to a new testing-method that would combine the advantages of both approaches. Moreover the approaches could benefit from each other. The model of the system that is created for the modelbased testing approach could also be used as an input for the testmanager, if it contained runtime constraints. This would lower the testing-costs for the testmanager. Additionaly the monitored data of the self-testing method could be stored and reused as test-data for later modelbased tests.

## References

[1] B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif. *Towards Testing Self-organizing, Adaptive Systems*, pages 180–185. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[2] G. Püschel, S. Götz, C. Wilke, and U. Aßmann. Towards systematic model-based testing of self-adaptive software. In *Proc. 5th Int. Conf. Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, pages 65–70. Citeseer, 2013. wrong paper in directory.

[3] G. Püschel, C. Piechnick, S. Götz, C. Seidl, S. Richly, and U. Aßmann. A black box validation strategy for self-adaptive systems. In *Proceedings of The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, S*, pages 111–116. Citeseer, 2014.