Testing self-adaptive systems - an overview

Tilo Werdin, Dominik Olwig

June 4, 2017

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 snowballing-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

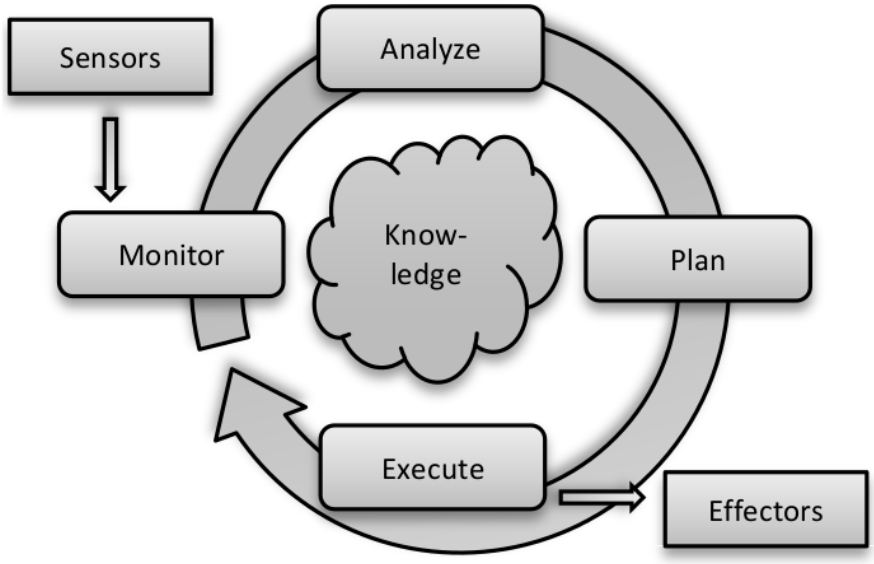

Figure 1: MAPE

2 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 the 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 stake-holders.

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.

There are already different alternatives that we want to discuss and compare with each other in this paper. On the one hand there is self-testing with different variations and on the other hand there is model-based testing.

missing: method (snowballing)

3 Method and result

3.1 Method

3.2 Self-Testing

The basic ideas of self-testing is to monitor the systems state at runtime and check 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 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.

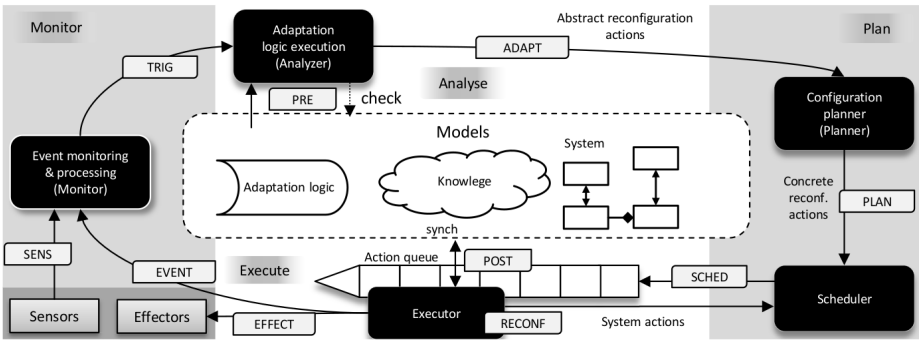

Figure 2: occurence of failures

3.2.2 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. [2] 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

In the basic idea the model-based testing can be compared to traditional unit tests that a testing engineer is writing to test software. But model-based testing is a further developement of unit testing. The idea of model-based testing is not to think of all different scenarios and writing tests to each one oneself, but to generate these tests. The target of generating tests for SAS is to touch all available scenarios and test every situation the SAS might be into.

3.3.1 finding failure scenarios

To find failure scenarios there is the need of searching for aspects that can fail during execution. Püschel et al. [3] defines 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 [4].

3.3.3 testing

After modelling different orthogonal state-flows or other behavior models, the main part of the generating test cases for the model-based testing is, merging the models together canonically and getting a huge set of test scenarios. For each of these test scenarios there will be created one special test. The great benefit is that the test engineer has not to think of every scenario himself but when the behavior is modeled, he will get every single test scenario from the generation from the models. So if there is any case that is very unlikely to happen, the testing engineer might leave it out or do not even think about it. But with this method it is tested anyway and if the system gets to this state there is a way to normalism.

4 Evaluation

4.1 Criteria

For our evaluation of the different testing strategies, we need some criteria. With the help of these criteria we can rate and compare the strategies.
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 encure 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

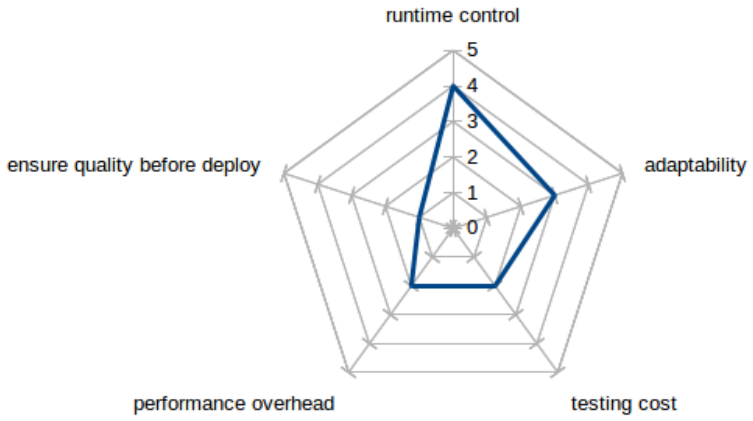

Figure 3: testmanager - Kiviat graph

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

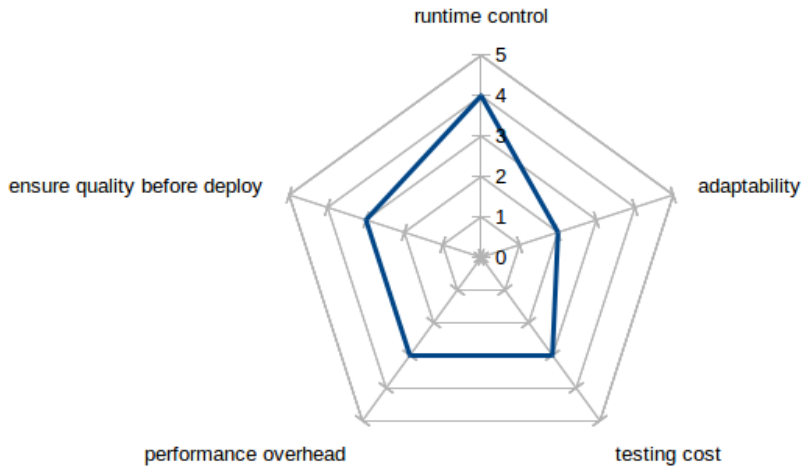

Figure 4: corridor enforcing infrastructure - Kiviat graph

+ An advantage of this method is, that testing the corridor enforcing infrastructure would be enough to guarantee good system behavior. The system itself does not need to be tested.
+ testing the CEI is easier than testing the system itself
- big infrastructure
- overhead at runtime

4.2.3 modelbased testing

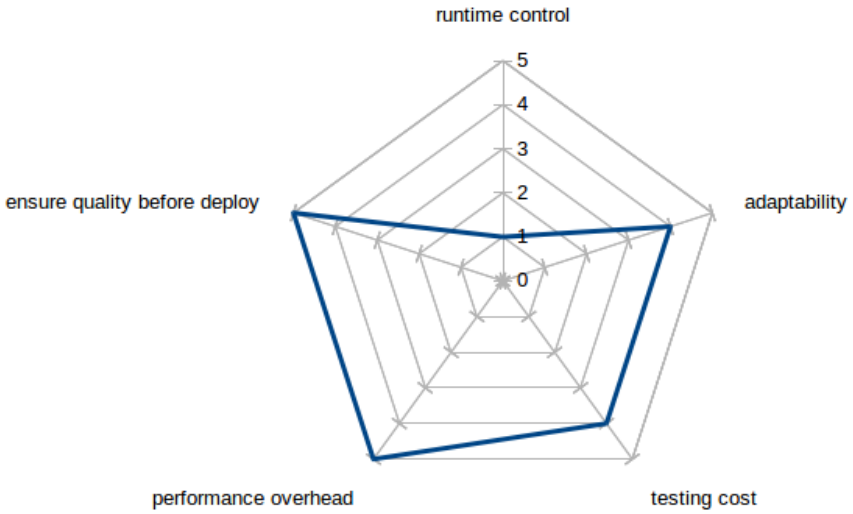

Figure 5: model based testing - Kiviat graph

5 Future Work

- our research has shown:
- modelbased testing good for ensure good behavior prior to deployment (needed for safety-critcal systems - certification process)
- self-testing good for dealing with adaptive character of SAS
-¿ maybe a combination of both methods can lead to better testing-methods
- how both methods could benefit from each other:
- modelbased testing -¿ creating a model of the system -¿ use this model including constraints as input for a testmanager, that can check during runtime
- self-testing methods -¿ monitoring system during runtime and collecting data -¿ keeping the data can improve offline-testing with model-based tests (cheap test-data)

References

[1] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

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

[3] 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.

[4] 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.