Testing self-adaptive systems - an overview

Tilo Werdin, Dominik Olwig

May 30, 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

2 Introduction

For engineering software there are several approaches. One that got more important over the past years is self-adaptive software for self-adaptive systems (SAS). A SAS is able to adjust to different situations in different places on its own. The great benefit and the cause of why the field of SAS is researched now is that the software engineer does not program all exact situations that the system might be in. The software engineer just models some situations and abstracts from that. The system itself then should find a way to behave in a good way.

This is done by control loops. A control loop in a SAS contains four main parts: monitoring, analysis, planning and execution (MAPE). As the name loop tells, 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 made.

The greatest thinkable application of SAS might be a human like robot that behaves completely autonomous. It would be designed by some system architect and programmed by some software engineer, but they will never imagine every single situation the robot might be in. The solution is, developing a system that adjusts on it's own and "learns" on it's own over the time. A smaller example could be an application that uses variable web services. In this application it shell not be important that a special service is reachable. The system chooses at runtime one of the services available and that could be very different ones each time the system runs.

Looking at the examples above, it is clear that these SAS need to be tested, in order to assure quality. This includes validation and verification of the systems. The worst case would be that there is a system produced that uses self-adaptation. It is delivered to the customer but was never tested and validated before. The customer starts the system and it is not working as expected and the system failure leads to emergencies. Therefore it is very important to test self-adaptive systems.

While SAS are constructed in an other way than "normal" systems and the process of developing SAS is quite different there is the need of new ways to test systems and their behavior. Traditional Unit-Tests will reach their limits as the testing engineer needs to think of every single test case on his own and as mentioned before it is nearly impossible to think of every single situation the SAS might be in.

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

missing: method (snowballing), motivation (overview) maybe to much detail in the first to paragraphs

3 Method and result

3.1 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.

3.1.1 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.

3.1.2 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.

3.2 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.

3.2.1 finding failure scenarios

To find failure scenarios there is the need of searching for aspects that can fail during execution. Pschel et al. defines quite a lot failure scenarios and explains them.

3.2.2 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.

3.2.3 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.

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

Kiviat-Graph of all methods...

information for evaluation:

selftesting:
+ keeping the monitored data can be a good source for later offline-tests
- overhead at runtime

test-manager
+ flexible, relatively independent component
- overhead at runtime (safe adaption with validation has higher time overhead, lower memory overhead; replication with validation has lower time overhead, but higher memory overhead)
- cannot guarantee good behavior prior to deployment -¿ need to be combined with traditional tests
- there might be situations, where the testmanager can not handle a validation of a constraint (e.g. response time)

corridor enforcing infrastructure:
+ 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

5 test

“hello world“
[3] [1]

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] T. M. King, D. Babich, J. Alava, P. J. Clarke, and R. Stevens. Towards self-testing in autonomic computing systems. In *Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07)*, pages 51–58, March 2007.