Adaptive Tests for Adaptive Systems: The Need for New Concepts in Testing for Future Software Systems

Benedikt Eberhardinger, Hella Seebach, André Reichstaller, Alexander Knapp, and Wolfgang Reif

Institute for Software & Systems Engineering, University of Augsburg, Germany

{ *eberhardinger, seebach, reichstaller, knapp, reif* } *@isse.de*

Abstract Software testing plays a major role for engineering future systems that become more and more adaptive to their environment. In order to fulfill the high demand, test automation is needed as a keystone. However, test automation, as it is used today, is counting on capture-and-replay-like scripting and is thus not able to keep up with intelligent systems. Therefore, we ask for an adaptive test automation and propose a model-based approach that enables self-awareness as well as awareness of the system under test which is used for automation of the test suites.

1 A Plea for Adaptive, Self-Aware Test Automation

With the growing adoption and resulting ubiquity of adaptive, autonomous software systems, be it in the Internet of Things or in Industry 4.0, quality assurance and testing of these systems becomes of increasing importance. The challenges of delimiting adaptation, integrating continual context changes, or handling learning and reasoning, to name just a few (see Siqueria et al.'s survey [1]), not to the least part affect *test automation* controlling test execution. When executing a test case, previous test effort may have caused adaptation or may have influenced the perceived context of the adaptive system.

We therefore propose that test automation for adaptive systems needs also to become autonomous and adaptive in order to adapt itself to the system under test (SuT). A prerequisite for adaptation and autonomy is to establish awareness, awareness of the test system, the test automation, as well as the SuT. This awareness could be used for autonomous decisions like which test case to be executed next or even to learn from test execution and the evolving SuT how to evolve the tests. For this purpose, test automation needs to be able to adapt, execute, and maintain itself to the ever-changing SuT.[1]

This paper, an extension of [2], includes the following contributions:

1. Self-aware test automation to adapting test strategies
2. Run time models to achieve awareness in test automation
3. A concept for test case diversity optimization

The paper is structured as follows: Section 2 introduces the Znn.com case study used throughout the paper. Within Sect. 3 we describe our approach for self-aware run time models in test automation implemented within the S# framework. Section 4 describes how the self-aware test automation enables an adaptive selection of logical test cases and how these are instantiated by a planner. The approach is incorporated into the related work in Sect. 5 and concluded in Sect. 6.

[1] These goals are adapted form the Keynote of Jeff Offutt at the ICTSS'16 in Graz, Austria, where he is also appealing to the community for more research in intelligent test automation.

2 Case Study: Znn.com

We use the Znn.com case study that has been widely established as the standard case study for self-adaptive systems. The following description of the case study is therefore an excerpt of Cheng [3] who defined the case study first. The Znn.com case study is an online service serving news content to its customers, like *cnn.com* or *sz.de*. Architecturally, Znn.com is a client-server system with a multi-tier architecture model. Znn.com uses a load balancer to equilibrate requests across a pool of servers, the size of which is dynamically adjusted. The business objectives at Znn.com are to serve news content to its customers within a reasonable response time range while keeping the costs of the server pool within its operating budget. From time to time, due to highly popular events, Znn.com experiences spikes in news requests that it cannot serve adequately, even at maximum pool size. To prevent unacceptable latencies, Znn.com opts to serve minimalist textual content during such peak times instead of providing its customers no service. The adaptation decision is determined by observations of overall average response time versus server load. Specifically, four adaptations are possible, and the choice depends not only on the conditions of the system, but also on business objectives: (1) switch the server content mode from multimedia to textual and (2) vise versa, (3) increase the server pool size, and (4) decrease the server pool size. Within the Znn.com case study, the self-adaption allows automation of adaptations that strikes a balance between multiple objectives.

For the proof of concept we implemented a simple version of the Znn.*com* case study in C#. The implemenatition incooperates a simple, adaptive load balancer which is based on the describtion of Cheng [3]. The evaluation setting runs on one single computer; the servers do not deliver real content but simulate different workload depending on what mode is active. The implementation of our approach runs within the S# framework (cf. Sect. 3.1) on the same computer as the Znn.*com* implementation.

3 Run Time Models for a Self-Aware Test Automation

Our approach is based on the concept of run time models. They are used to reflect the actual state of the SuT and of the test system in order to establish a self-aware test automation. Based on the awareness the test automation is able to adapt itself in an autonomous way by a reasoner and instantiate new test cases by a planner, as shown in Sect. 4. Within this approach we integrate two different possible test strategies that build the foundation of the adaptation and the autonomy of the test automation. The first strategy is based on operational profiles used for generating continuous test input for the SuT by simulating the environment. The second strategy is a fault-based test strategy where test cases are defined by environment faults that cause the actual system for action. Both strategies have been proven to be effective in revealing failures within autonomous software systems (cf. [4] for the first and [5] for the latter) and are now combined for test automation for end-to-end tests. We show how the S# modeling framework is used to build run time models for the self-aware test automation which is combining of these to strategies.

3.1 The S# Framework: Executable Models

The S# framework incorporates an integrated, tool-supported approach for modeling and analyzing component-oriented systems [6]. It provides a component-oriented *domain specific language* embedded into the C# programming language. S# inherits all of C#'s language features and expressiveness, including all state-of-the-art code editing and debugging features provided by the Visual Studio development environment. It is the foundation for our model-based approach for test automation since both of the used test models are implemented in S#. Listing 1 shows an excerpt of the model of the server in the ZNN.com case study. The model describes the internal state of a server as well as further information that are used in the test automation. The faults are representing logical test cases described in the model by overwriting the actual functionality. Note that in this example, for the sake of simplicity, the actual implementation is also included in the model as C# code. For instance, the `Activate` function which sets the `_isServerActive` attribute to `true` could be overwritten by the fault implementation which is empty and does not change the state of the attribute, similar for the `AddQueries` function. The `CannotExecuteQueries` fault is annotated by its activation criterion that states that this fault should be active if the proxy has few servers available. Further, the selected server is set to `auto` which means if a set of server instances matches to this criterion the planner within the test automation will determine which and how many of the set of servers should be selected for the fault activation. The actual property that is

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

  public void Activate()    { _isServerActive = true; }
  public void AddQueries(List<Query> queriesToExecute) {
    _executingQueries.AddRange(queriesToExecute);
  }
  [Transient] class ServerCannotActivate : Fault {
    public void Activate() { }
  }
  [Activation("TooFewServers", selectedServer="auto")]
  [Persistent] class CannotExecuteQueries : Fault {
    public void AddQueries(List<Query> queriesToExecute) { }
  }
  /* ... */
}
```

Listing 1: Partial S# component representing a server of ZNN.com case study.

linked by the activation criterion is not shown here. It is formulated as a boolean function in S#. Thus, the fault is a logical test case with many different possible concrete test cases that could be executed by the test automation. The planning module is helping by this process.

3.2 Building and Using Run Time Model in Test Automation

A model is a simplified effigy of the reality. The simplification is achieved by abstraction resp. reduction of the reality with a certain pragmatism. We use the models in our approach to gain a gray-box view of the SuT. Therefore, we seek for an appropriate representation that enables selection, adaption, combination, ordering, execution, and evaluation of tests based on the current state of the system. The model is used to easily query the current state of the SuT and its environment, but also to query the current state in order to evaluate it with a given online oracle (for a detailed description of the used online oracle see [7, 4, 5]). Further, the models are not limited to query the current state they are also used to execute tests, i.e., the models have to be executable in order to stimulate the SuT as described by the tests to be executed. We use a multistage strategy that first queries the current state, uses that information in the rule-based reasoning to gain tests to execute, executes the tests, updates the model at run time, and evaluates the state of the SuT by applying the oracle on the model. The main requirements for the model that we derive are, that the model has to represent the current state of the SuT in an abstract manner, the models need to be executable, and the models needs to provide sufficient information for reasoning and evaluation by the oracle. For this reason, we use executable run time models, provided by S#. The models consist of a static description of the SuT, whereas, the static description is instantiated and continuously updated by the current state of the SuT. In our ZNN.com case study the client is a component of the context model and the server pool of the SuT model. Mapping the actual state of the SuT in this context incorporates, among other things, to update the current servers within the pool. This information of the model is used by the reasoner

to check whether a certain number of servers has been added or removed to the server group that triggers a rule that activates a test. The models of the context and the SuT differ: the context model also includes dynamic information that enables it to execute different usage profiles of the test suite. In the case of the client a certain instance has a state machine with states such as *idle*, *requesting*, and *waiting*, that are selected according a profile that defines a probability of switching between the different states. The gain is a simulation of the environment that is executed on the SuT; the evolvement of the SuT enables it to execute the situational aware tests that activate faults.

4 Adaptive Test Execution @RunTime

The main constituent of our test automation is the adaptive test execution which is based on the self-awareness originated from the run time model established in the section before. Two main test strategies the execution is relying on: (1) a simulation of the SuT's environment and (2) an interface for activating component faults in the controlled environment of the SuT. The first strategy enables to continuously trigger the SuT and keep it operating and possibly evolving. This is the basis for strategy (2) where a given set of test cases is executed by activating component faults of the controlled enviroment which is an established technique for testing autonomous systems (cf. [5]). In this way it is possible to define different adaptation rules for the test execution within the model that trigger which of the anticipated behaviors of the environment to be simulated and which fault to be activated. The first set concerning the simulated environment is composed of probability functions that map a probability to a transition from one environment state to another that is formally described as a Markov chain. An example for the relevant environment in the case study is the client requesting content. The S# model of the client therefor includes a state machine with the states *idle*, *requesting text*, and *requesting media* where it is possible to get from each state to every other and each transition has a certain probability, gained from user surveys. The second set contains the injected faults that might be activated under some conditions. It is based on faults or set of faults modeled in the components of the S# model. Considering the example of Listing 1 different kinds of faults are shown: persistent faults and transient faults. The first kind of fault indicates that once the fault is activated it remains active for the rest of the execution whereas the later one might be deactivated. Note that in the test automation we expect on the one hand side that a model of the SuT's environment is given by a test engineer that is used to generate continual test input and on the other hand we expect explicit test cases with an activation criterion designed by a test engineer. The test automation is using these two inputs as a test suite for the adaptive test execution.

4.1 Rule-Based Reasoning for Adaptive Test Execution

The adaption is enabled by the awareness of the current state by the test model and is used by a rule-based reasoning engine which makes the execution adaptive. Thus, it is possible to evaluate the current state of the test system and the SuT and select an appropriated test case for execution in the current situation. As described before, we use two strategies. The simulation of the environment is driven by an probabilistic Markov model of the environment which is used to describe the environment's common behavior. The faults instead are activated based on situational patterns, the activation criteria, evaluated by the reasoner. The activation criteria are described in form of constraints that are annotated to the tests, like shown in listing 1; it could be said: the tests know their purpose. Based on the sets of rules it is possible to reason at run time over the current state of the model and select the next test step based on the rules defined. Another important component in the framework is the planner that enables to enhance the test suite at run time.

4.2 Planning Optimal Rule Instantiations

Depending on the chosen system configuration, a particular rule may subsume quite a number of concrete implementations. The question, which of them to choose, i.e., which test step to execute if there are many, resembles the challenge of instantiating logical test cases by concrete ones in traditional testing. Let us, for instance, consider the rule which introduces the persistent fault "CannotExecuteQueries" in listing 1. At test execution, this fault needs to be instantiated for one out of all the servers which are deployed in the considered configuration. As typically huge environmental state spaces prevent us from demanding manual solutions for resulting decision processes from the tester, we propose to equip the rule-based reasoner with a planning module that is able to automatically solve this job. In the aforementioned example this module is activated by the keyword "auto" at activation "TooFewServers". This marks the choice of which server to be affected as being non-deterministic and thus to be decided by the planning module.

Parametrized with some goals, which we assume to be given by the test engineer, the planning module uses the executable models to search for optimal decisions concerning the rule instantiations. An exemplary goal might be a kind of *action diversity*, which we see as a counterpart of code coverage criteria in the context of proactive, adaptive systems. This test indicator, based on the behavioral distance metric introduced in [8] and [9], can be used to measure the difference between the system traces which are expected to be triggered by particular rule instantiations. Maximizing the action diversity thus means to maximize the difference of

triggered traces. As shown in [8], the underlying behavioral distance function can be learned on-the-fly by simply observing the system under test in simulation. This process can be directly integrated within our S#-models. Given the function, the planning module tries to find those traces which maximize the distance to the traces that were already seen. Note that the planner module is not limited to our notion of diversity but is generally able to cope with arbitrary quantifiable goals.

5 Related Work

Our approach extends current test automation concepts by a notion of adaptiveness (w.r.t. system and context states). Self-aware test models enable us to design a new kind of automatable test cases that incorporate situational aspects, purpose, but also information about the correct system state.

As can be seen in [10], current approaches primarily execute test scripts, mostly without any context description, or replay captured scenarios that have been recorded through manual testing. Though they strive to optimize effectiveness in test execution, this is often at the cost of reactiveness and adaptiveness in regards to changing context or system states. There is, however, a need for reactive tests – especially when dealing with systems that are able to change their internal structure in response to contextual changes for themselves; or in other words: *adaptive systems need adaptive tests*. We solve this by using models as run-time reflection of the current state. This approach is inspired by the model@runtime community (cf. Bencomo et al. [11]), even if they are rather concerned with adapting system strategies instead of testing.

Existing approaches for testing adaptive systems are focused on test case generation and the usage of the test output to tweak the system performance (cf. Siqueira et al. [1]). The concrete automation of tests was previously only done for dedicated test cases, not in a general approach as we propose it. Consequently, we propose a new and thorough approach for intelligent test automation that is especially needed for autonomous system testing.

6 Conclusion & Outlook

In order to cope with future software systems, test automation needs to become self-aware, adaptive, and evolving in its execution. In present work, we approached these requirements building on run-time models of the SuT and its context. In future work, we will evaluate our findings and concepts in a bigger, industrial case study. Therefore, we prepare an Apache Hadoop cluster to test in a Docker environment that uses an adaptive resource manager. The evaluation will take place in a distributed setting. Beside that, we are going to extend the planning module with abilities to plan even new fault-based test cases for execution.

Acknowledgment This research is sponsored by the research project *Testing Self-Organizing, Adaptive Systems (TeSOS)* of the German Research Foundation.

References

[1] B. R. Siqueira, F. C. Ferrari, M. A. Serikawa, R. Menotti, and V. V. de Camargo, "Characterisation of challenges for testing of adaptive systems," in *Proc. 1st Brazilian Symp. on Systematic and Automated Software Testing*, SAST, pp. 11:1–11:10, ACM, 2016.

[2] B. Eberhardinger, A. Habermaier, and W. Reif, "Toward adaptive, self-aware test automation," in *Proc. 12th IEEE/ACM Int. Wsh. Automation of Software Testing, AST@ICSE 2017*, pp. 34–37, 2017.

[3] S.-W. Cheng, *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, CMU, 2008.

[4] B. Eberhardinger, G. Anders, H. Seebach, F. Siefert, A. Knapp, and W. Reif, "An approach for isolated testing of self-organization algorithms," *CoRR*, vol. abs/1606.02442, 2016.

[5] B. Eberhardinger, A. Habermaier, H. Seebach, and W. Reif, "Back-to-back testing of self-organization mechanisms," in *Proc. 28th IFIP Int. Conf. Testing Software and Systems (ICTSS)*, Springer, 2016.

[6] A. Habermaier, J. Leupolz, and W. Reif, "Executable Specifications of Safety-Critical Systems with S#," in *Proc. of DCDS*, IFAC, 2015.

[7] B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif, "Towards testing self-organizing, adaptive systems," in *Proc. 26th IFIP Int. Conf. Testing Software and Systems (ICTSS)*, vol. 8763 of *LNCS*, Springer, 2014.

[8] A. Reichstaller and A. Knapp, "Transferring context-dependent test inputs," in *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pp. 65–72, IEEE, 2017.

[9] A. Reichstaller and A. Knapp, "Compressing uniform test suites using variational autoencoders," in *IEEE Int. Conf. Software Quality, Reliability and Security Companion (QRS-C)*, pp. 435–440, 2017.

[10] M. Polo, P. Reales, M. Piattini, and C. Ebert, "Test Automation," *IEEE Software*, vol. 30, no. 1, pp. 84–89, 2013.

[11] N. Bencomo, R. B. France, B. H. C. Cheng, and U. Aßmann, eds., *Models@run.time - Foundations, Applications, and Roadmaps*, vol. 8378 of *LNCS*, Springer, 2014.