OPEN ACCESS
OPUS AUGSBURG

Universität Augsburg
Universitätsbibliothek

Toward adaptive, self-aware test automation

Benedikt Eberhardinger, Axel Habermaier, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Eberhardinger, Benedikt, Axel Habermaier, and Wolfgang Reif. 2017. "Toward adaptive, self-aware test automation." In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST), 20-21 May 2017, Buenos Aires, Argentina*, edited by Hong Zhu, Junhua Ding, Patricia Machado, and Marc Roper, 34–37. Piscataway, NJ: IEEE. https://doi.org/10.1109/ast.2017.1.

Nutzungsbedingungen / Terms of use: **licgercopyright**

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/

Toward Adaptive, Self-Aware Test Automation

Benedikt Eberhardinger, Axel Habermaier, and Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, Germany
E-Mail: {eberhardinger, habermaier, reif}@isse.de

***Abstract*—Software testing plays a major role for engineering future systems that become more and more ubiquitous and also more critical for every days life. 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 scripting and capture-and-replay and is not able to keep up with autonomous and intelligent systems. Therefore, we ask for an adaptive and autonomous 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.**

I. A Plea for Self-Aware Test Automation

Things are changing in software engineering, many different movements are increasing the complexity as well as the importance of software. Software percolates the most critical industries and every day's life, this development is named by the internet of things, industry 4.0, amongst others. This recent development changes the requirements to the engineering procedure of software, since software is crucial in critical areas, software is more complex, and software is becoming more autonomous. Software testing plays a major role for engineering these kinds of new systems. Without a suited test engineering approach it is not possible to address the high quality standards needed nor to solve the complex tasks demanded. Different approaches are addressing these challenges (the survey of Siqueria et al. [1] gives a good overview of the most recent ones) on different levels. However, similar to classical testing test automation is neglected by the community especially when it comes to integration and end-to-end testing. As a matter of course, there are commonly used and well-tried approaches like JUnit or NUnit that implement test automation for single functions or classes, however, the automation takes a bunch of human invention for creation, execution, and maintainability. It is even getting worse when it comes to end-to-end testing that is mostly carried out by capture-and-replay tools, e.g., with Selenium. For testing autonomous systems using these test automation tools, that is an uneven struggle: test automation needs also to become autonomous and adaptive in order to adapt itself to the system under test (SuT). That encompasses to be self-aware (i.e., to know the purpose and context of test cases), to be aware of the SuT, and to use that awareness for decisions, like what test case to be executed next. For this purpose, the test automation needs to be able to adapt, execute, and maintain itself to the SuT.[1]

[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.

This position paper includes the following contribution:

1) Self-aware test automation to adapting test strategies
2) Run time models to achieve awareness in test automation
3) Proof of concept applied to a cloud application

The paper is structured as follows: started by introducing of the ZNN.com case study in section II that is used as a running example as well as for the first proof of concept in section V. Within section III we describe our approach that makes use of the concepts of run time models, implemented within the S# framework. The approach is incorporated into the related work in section IV and concluded in section V.

II. 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 [2] that defined the case study first. The ZNN.com cases study is an online service serving news content to its customers, like cnn.com or nyt.com. 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.

III. Run Time Models for a Self-Aware Test Automation

For our approach we combine several techniques and concepts together in order to form a self-aware and adaptive test automation framework. On the one hand side, a run time model of the actual SuT is used in order to reason about the SuT and enable the tests to gain knowledge needed for their execution

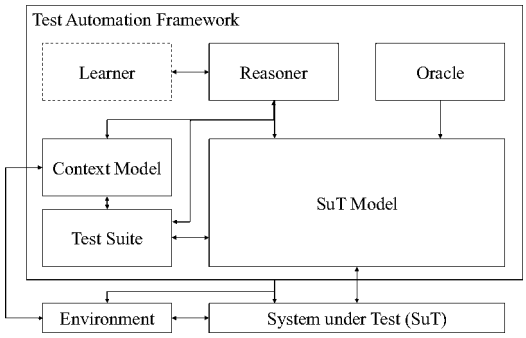

Fig. 1. The test automation architecture consits of two models: one for the SuT and one for the context (resp. the environment) that enable it to reason about its state (representing a instance of the the SuT and its environment) and the test suite to adaptively execute tests by the reasoner and evaluate them based on the oracle component.

(or even the decision whether to run or not). On the other hand, we integrate two different possible test strategies that give the test automation the necessary degrees of freedom for adapting itself. The test model is used for the decision which test case to be executed or more generally which action of the test automation to be taken. This decision is possible since different test cases and even strategies are defined within the test automation. Thus, it is possible to burst the static structure of classical test automation. Both strategies have been proven to be effective in revealing failures within autonomous software systems (cf. [3] for the first and [4] for the latter) and are now combined for a test automation for end-to-end tests.

Next we are diving into greater detail on the underlying modeling framework of S# (section III-A). We show how to build and use the run time models of the SuT and the environment (section III-B) and how the reasoning is performed within different possible strategies (section III-C). Last we explain how a automated oracle is incorporated (section III-D).

A. The S# Framework: Executable Models

Our modeling framework, S#, incorporates an integrated, tool-supported approach for modeling and analyzing component-oriented systems [5]. S# 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 (cf. fig. 1) 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, its provided ports for other components, as well as further information that are used in the test automation. One of the faults is annotated by its activation criterion that states that this fault should be active if the proxy has less than two servers activated. S# models are *executable*, allowing them to be simulated, tested, visualized, and debugged in addition to automatically reasoning about the model and its current state.

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

   public void Activate()    { _isServerActive = true; }
   public void AddQueries(List<Query> queriesToExecute)
   {
         _executingQueries.AddRange(queriesToExecute);
   }
   [Transient] class ServerCannotActivate : Fault {
      public void Activate() { }
   }
   [Activation("_connectedProxy._activeServers.length<2")]
    [Persistent] class CannotExecuteQueries : Fault {
      public void AddQueries(List<Query> queriesToExecute) {}
   }
   /* ... */
}
```

Listing 1. Partial S# component representing a server of ZNN.com case study, showing some of the internal state, provided ports, and two of the server's faults where on is annotated by its activation criterion. Other internal state, the subcomponents, and the state machine describing the server's behavior are omitted due to space restrictions.

The underlying model of computation is a series of discrete system steps, where each step takes the same amount of time that is important for the simulation abilities of the models. Structural and behavioral design variants can be modeled using the modularity and composability concepts of S#'s modeling language, which is most useful when analyzing the changing model of the evolving system at run time.

B. 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 in order 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 oracle. Further, the models are not limited to querying 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, use that information in the rule-based reasoning to gain tests to execute, execute the tests, update the model at run time, and evaluate the state of the SuT by applying the oracle on the model. The main requirements for the model that we derived are, that the model has to represent the current state of the SuT in an abstract manner, the models need to be executable, 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*, among others, that are selected according the 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.

C. Test Selection at Run Time by Rule-Based Reasoning

In our approach the test execution within the test automation is based on a simulation of the environment and injecting faults in the controlled environment of the SuT. These are the two main test strategies that are selected by a rule-based reasoning approach. We are able to define different adaptation rules within the model that on the one hand side, 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.

In the first version of the reasoner, presented in this paper, the simulation of the environment is probabilistic, i.e., the probabilities in the model are used to simulate the environments common behavior. The faults instead are activated based on situational patterns evaluated by the reasoner. The situational patterns are described in forms of constraints that are annotated to the tests in the test suite, like shown in listing 1; it could be said: the tests know their purpose. Based on the sets of rules we are able to reason at run time over the current state of the model and select the next test step based on the rules. Thus, the ability of the test automation to adapt is in large parts within the reasoner and is based on the awareness of the model about its state and the state of the SuT and its environment. A component of future work in the framework is the sketched learner in fig. 1 that enables to enhance, extend, or delete rules at run time. Thus, we propose for adaptive and self-aware test automation to enrich the tests by information how, why, and when they should be executed in order to use the information at run time for reasoning.

D. Incorporating a Constraint-Based, Automated Oracle

Another gain of the the run time model, despite using it for reasoning and executing it, is the ability to evaluate the current state of the system by an constraint-based oracle on it. If the mapping of the current state of the system to the model is completed the constraints, defining the correct behavior, can be evaluated fully automatically. As we have already shown in previous work in [3], [6], [4] a constraint-based description of the oracle can be used very effectively for describing the intended behavior of self-adaptive, autonomous systems. The challenge is the mapping between model and SuT; as a state in the model is discrete and within the SuT continuously. We use the step-wise execution model of S#, with a micro-/macro-step semantic, with the steps described before. After execution is finished, we use the current snapshot of the system by sensing at that point in time the state. Of course, this first approach is prone to missing states and combining values of minimal different points in time to one discrete state. However, by the following assumptions we are able to use this concept for our test automation: First, the system is not supposed to change its state very fast within milliseconds in a way that it will affect the state of the model, e.g. the number of servers active will not change more than once from on millisecond to another. Thus, we are able to benefit from the abstraction made in the model here. Second, if the system violates a constraint and a millisecond later it fulfills it again, that is not a failure at all that we want to reveal. For adaptive systems, that are able to recover from faulty situations, a temporary failure is acceptable, if it is able to recover itself within a reconfiguration. In the ongoing work, we will put more effort in this mapping and trying to verify these assumptions.

IV. Related Work

Our approach is extending the current state of the art of test automation by making test automation self-aware and adaptive, i.e., we provide a concept to enrich test automation by reasoning and learning of the current, the past, and the future state of the SuT. The gained knowledge enables to design a new sort of automatable test cases that incorporate situational aspects, purpose, but also information about the correct system state. Polo et al. [7] provide an overview of current test automation methodologies and technologies where it could be seen that current approaches foremost are concentrate on executing test scripts without any context or replaying captured scenarios that have been recorded through manual testing. The approaches have a great capability in fast and efficient execution, but lack in maintainability and have to be reworked after changes in the system and further have to be governed with high effort. Especially autonomous, self-adaptive systems demand a higher degree of intelligence within test automation, or in other words: adaptive systems need adaptive tests. We therefore build up on concepts of the model@runtime community (cf. Bencomo et al. [8]) and use models as a run time reflection of the current system, but in our case we use this information within the test automation not for adapting the systems strategies. Our decision for the

implementation of that concept, using the .NET run time and type system as its meta-metamodel in S# might sound unconventional, however, as .NET was specifically designed for efficient program execution and run time object composition, both of which form the basis of S#s support for run time model adaptation. The approaches for testing adaptive system could be clustered into approaches that are performed alongside the execution of the rolled out system and approaches that are performed at the design time of the system; both have identified non-determinism and the emergent behavior as the main challenges for testing adaptive systems. For testing alongside the deployed system, the approaches take up the paradigm of run time verification introduced by Leucker and Schallhart [9]. For instance, Camara et al. [10] uses these concepts in their approaches on testing the non-functional properties of resilience of the adaptive system. In our testing approach, by contrast, we are not limited to non-functional requirements. Further, we concentrate less on test case generation, but more on their automation. Still, we also use the basic concepts of run time testing. Nguyen et al. [11] as well as Zhang et al. [12] test their systems in a classical manner during development. Both approaches consider some dedicated parts of the system and are mainly focused on test case generation. Nguyen [11] promotes an approach for a component test suite, but does not consider interactions between or organization of components. All approaches of testing adaptive and autonomous systems are focused on test case generation and the usage of the test output to tweak the performance of the system (cf. Siqueira et al. [1]). The concrete automation of the tests within the related work is done only for dedicated test cases, not in a general approach as we propose it that further uses data from the executed system to enrich them. Consequently, we propose a new and thorough approach for intelligent test automation that is especially needed for autonomous system testing.

V. Proof of Concept & Outlook

For the proof of concept we implemented a simple version of the ZNN.*com* case study. The implementation incorporates a S# based implementation of a simple, adaptive load balancer and servers that are able to switch between modes (the media and the text mode). The evaluation setting is not distributed and 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 the described test automation approach runs within the S# framework and on the same computer as the ZNN.*com* implementation. The first proof of concept, that incorporates the implementation of the concept to the ZNN.*com* case study, shows that the concept is able to be mapped to a scenario that is omnipresent in today's world (cf. similar industrial applications like amazon cloud services, google web services, etc.). Furthermore, we are convinced that other industrial applications, e.g., autonomous production systems or autonomous energy systems, are also applications that can be tested with our approach. This confidence is based on our previous work where we developed approaches for generating, executing, and evaluating tests in this application domains based on similar concepts (cf. [3], [4], [6]). Parts of the applications are also partially already modeled in S# and we are working on proving our assumptions. Furthermore, the next steps in our work cover the implementation of the test automation framework on an industrial cloud application and evaluate the abilities of the approach. Indeed, the real need for adaptive and self-aware test automation will be made more obvious in bigger, industrial applications. However, we observed situations where the adaption payed off in our proof of concept: first, the fault-based test cases have been activated in much more situations than we anticipated beforehand, and second, during the extension or replacement of the adaption strategy we did not need any maintenance of the test setting. During the implementation, we revealed different failures, e.g., in a very volatile setting of the environment the adaption mechanisms missed to refresh the current status of the servers which leads to a inconsistency.

Overall, the proof of concept showed that the approach is sustainable and promising in its results. We are going to extend the evaluation and the evaluation case toward industrial applications. Further, we will extend the reasoner by a learner that is able to add, remove, or modify the rule set in order to be more adaptive to the current state of the SuT.

Acknowledgment

This research is sponsored by the research projects *Testing Self-Organizing, Adaptive Systems (TeSOS)* and the research unit OC-Trust (FOR 1085) 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. of the 1st Brazilian Symp. on Systematic and Automated Software Testing*, ser. SAST. ACM, 2016, pp. 11:1–11:10.

[2] S.-W. Cheng, "Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation," Ph.D. dissertation, CMU, 2008.

[3] 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)*, ser. LNCS. Springer, 2014, vol. 8763.

[4] 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.

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

[6] 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. [Online]. Available: http://arxiv.org/abs/1606.02442

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

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

[9] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, 2009.

[10] J. Cámara and R. de Lemos, "Evaluation of resilience in self-adaptive systems using probabilistic model-checking," in *Proc. 7th Int. Symp. Software Eng. for Adaptive and Self-Managing Systems (SEAMS)*, 2012.

[11] C. D. Nguyen, "Testing techniques for software agents," Ph.D. dissertation, Uni. di Trento, 2009.

[12] Z. Zhang, J. Thangarajah, and L. Padgham, "Model based testing for agent systems," in *Proc. 8th Int. Conf. Autonomous Agents and Multiagent Systems (AAMAS)*, 2009, pp. 1333–1334.