Case study: adaptive test automation for testing an adaptive Hadoop resource manager

Benedikt Eberhardinger, Hella Ponsar, Gerald Siegert, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Eberhardinger, Benedikt, Hella Ponsar, Gerald Siegert, and Wolfgang Reif. 2018. "Case study: adaptive test automation for testing an adaptive Hadoop resource manager." In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), 16-20 July 2018, Lisbon, Portugal*, 513–18. Piscataway, NJ: IEEE. https://doi.org/10.1109/QRS-C.2018.00092.

Case Study: Adaptive Test Automation for Testing an Adaptive Hadoop Resource Manager

Benedikt Eberhardinger, Hella Ponsar, Gerald Siegert, and Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, Germany
Email: { eberhardinger, ponsar, reif } @isse.de
gerald.siegert@student.uni-augsburg.de

***Abstract*—Coping with adaptive software systems is one of the key challenges testing is currently faced with. In our previous work, we proposed to enable the test system itself to be adaptive to the system under test as a solution. The adaptation is built up on the concepts of a self-aware test automation enabling to use this information to sequence, instantiate, or update the test suite to the current situation. In our test framework the modeling language S# allows to use a run-time model to do so in a model-based testing approach. In this paper, we demonstrate how our concepts of adaptive, self-aware test automation are applied to a real world scenario: testing an adaptive resource manager of Hadoop. We show the steps necessary to implement the approach and discuss our experiences in this case study paper.**

I. Adaptive Tests for Adaptive Software Systems

Adaptive software systems are characterized by their continued life cycle after the system's development and initial setup [1]. According to Salehie & Tahvildari [1] the life cycle is continued in order to evaluate the system and respond to changes at all time. Consequently, it is possible to deal with an ever-changing environment of the software system. This flexibility and resilience is needed and exploited in order to overcome the increasing complexity of nowadays software systems. Different movements, e.g., the Internet of Things, Industry 4.0, amongst others, that percolate critical industry and consequently everyday's life demand the concepts of adaptation. Consequently, effort has been put in the development and research for engineering these kind of system, cf. de Lemos et al. [2]. Quality assurance is a crucial part of the overall software engineering process which is addressed on different level, as encompassed by the state of the art survey collection in the recent book [3]. We consider thorough software testing as a promising way to address the needs for high quality of adaptive software systems. One particular challenge that we identified is the need for a powerful test automation in order to cope with the complex system under test (SuT). As a matter of course, there are commonly used and well-tried approaches for test automation 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). Thus, a test automation needs 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 execute and maintain itself as well as to adapt to the SuT. In [4], [5] we introduced an approach for an adaptive test automation suited for adaptive software systems. The concept builds upon an executable-modeling paradigm of the S# modeling language that enables to reflect the current state of the SuT onto the model derived from the state of the environment and the SuT that makes it possible that the test cases use the knowledge to adapt their strategies (controlled by a reasoning engine). A test therefore has a situational pattern that describes when it might be executed, i.e., the test knows its purpose.

In order to demonstrate the abilities of the approach we chose the *Hadoop System*[1] which is equipped with an adaptive resource manager, responsible for task scheduling and allocation, as described by Zhang et al. [6], as an application case. The application is set up in a *docker*[2] environment using a desktop computing grid. Within this paper, we demonstrate how our approach of adaptive test automation applies to this complex real world application. Leading to the following contribution delivered in this paper:

1) A case study is given for adaptive testing of a distributed, adaptive real world software systems.
2) An S# test model for an adaptive Hadoop system is provided for adaptive test automation.
3) An implementation of a test harness for distributed Hadoop system connecting S#.
4) A run-time of our S# model reflecting a distributed *Hadoop System* in our S# environment.

The reminder of the paper is organized as follows: Section II introduces our adaptive test automation framework, that is applied to test an adaptive Resource Manager of Hadoop which is described in section III. In section IV, we show how the S# test model is build for the case study and in section V the test driver implementation for the adaptive test automation of Hadoop is explained. The experiences gained by the application of our approach to an adaptive Hadoop system

[1] http://hadoop.apache.org
[2] https://www.docker.com

is summarized in section VI.

II. An Adaptive Test Automation Framework in S#

In our previous work [4], [5], we developed an adaptive test framework in the S# modeling language. The framework is based on a run-time model of the SuT and its environment that enables a self-ware and consequently adaptive test automation framework. The focus of the framework is to supply a model-based approach to specify a test suite with information about its intention in order to allow for adaptation of the test execution at run-time. In order to enable the decision making we described a rule-based reasoning approach in [4] as well as planner based approach in [5]. Both need no further customization to the case study and can be directly applied on the executable model of the SuT. The executable model is key element that needs to be build for the SuT. It builds the foundation for the oracle that can be defined as well as the test suite to be integrated.

A. The S# Framework: Executable Models

Our modeling framework, S#, incorporates an integrated, tool-supported approach for modeling and analyzing component-oriented systems. Its models are *executable*, allowing them to be simulated, tested, visualized, and debugged in addition to automatically reason about the model and its current state. 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. S# provides a component-oriented *domain specific language* embedded into the C# programming language. In other words, S# models are *represented* as C# programs; conceptually, however, these programs are still models. Even those parts of S# models that do in fact represent software components are not intended to be used as the actual implementations of the real software: the models are usually an abstraction of the real software's behavior in order to reduce the complexity of the model. S# inherits all of C#'s language features and expressiveness. Every .NET library and tool can be used, including all state-of-the-art code editing and debugging features provided by the Visual Studio development environment.

B. Incorporating a Constraint-Based, Automated Oracle

Another gain of the run-time model, despite using it for reasoning and executing, is the ability to evaluate the current state of the system by a constraint-based oracle. 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 [7], [8] a constraint-based description of the oracle can be used very effectively for describing the intended behavior of self-adaptive, autonomous systems. Thus, the challenge is here the mapping between model and SuT; whereas, a state in the model is discrete and within the SuT continuously. Therefore, we use the step-wise execution model of S#, with a micro-/macro-step semantic. 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. Since this reconfiguration is longer than the inaccuracy of the measurement, we are fine here too. In the ongoing work, we will put more effort in this mapping and trying to verify these assumptions.

III. An Adaptive Resource Manager for Hadoop

Hadoop is one of the most popular and wide-used software platforms for big data processing and for using the MapReduce paradigm to a large number of different applications and workloads. The gain of its application depends on the configuration of a bunch of parameters which need to be tuned for a specific task or workload. The *YARN* (*Yet Another Resource Negotiator*) resource manager is the component of Hadoop which is responsible for scheduling and controlling the workload within the cluster. The parameterization of *YARN* is decisive for the job performance. The best practice for setting the parameter is a best-effort configuration that is based on experience or static profiling, relying on apriori knowledge about the job. Zhang et al. [6] developed a self-adaptive component on top of *YARN*. It is an implementation of the *MAPE* architecture (cf. [9]), i.e., a control loop that *m*easures, *a*nalyzes, *p*lans, and *e*xecutes adaptation of the parameter setting of *YARN*. Zhang et al. [6] showed that they are able to speedup the Hadoop instance up to 40% in a volatile environment compared to the best effort solution. We use the implementation of [10] which implements the concepts of [6]. We deployed the implementation in a docker-swarm that uses two desktop computers equipped with Intel i5-4690 processors with 4 cores, 16 GB RAM, 512GB SSD and Ubuntu 16.04 LTS as OS. The concrete deployment configuration can be seen in fig. 1. A prerequisite for testing is to have a set of testable requirements. For the sake of simplicity, we only used a subset of the overall requirements that can be extracted from the Hadoop documentation as well as the additional requirements of the adaptive extension documented in [6], [10]. Nevertheless, still the full power of the framework can be shown by using this subset. Since our test automation [4], [5] is focused on functional testing only functional requirements of

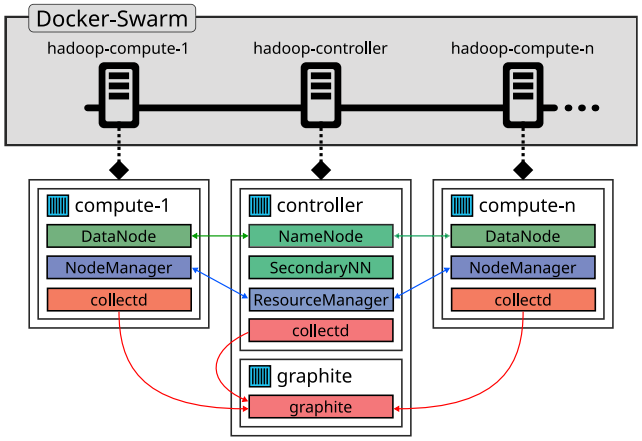

Fig. 1. Docker-swarm based deployment architecture of Hadoop instance with the adaptive extension by Zhang et al. [6], according to [10].

the *YARN* application are considered. The following functional requirements are used in our case study and are implemented in the automated oracle:

- A task will be completed, if it is not canceled
- No workload is allocated to inactive, defected, or disconnected nodes
- Parameters of the configuration are updated by the adaptation loop, if a certain rule applies
- Defects or disconnections are recognized

IV. S# Test Model for Hadoop Test Automation

A first step is to build the model in our modeling language S#. The model is used for the whole test process, i.e., the input generation, test execution, test evaluation, and the judgment. The overall model consists of a static and a dynamic part. Whereas the static part describes the structure of the SuT and its related constraints, the dynamic part is responsible for the adaptive automation of the test suite. As the model is executable it further incorporates the test driver.

A. Static Test Model

The static test model describes the components of the SuT, i.e., the *YARN* component, as a domain model and the requirements to be tested as a constraint-based oracle. The model-based testing paradigm pays off in this large scale industrial case study due to its abstraction abilities making our approach scalable. For this purpose, the model must be focused on the test purpose, that is defined by the set of investigated requirements outlined before.

1) Domain Test Model: The domain test model focuses on the test components of the SuT and their relations. Figure 2 shows the graphical representation of the classes that build the domain test model in S#. In general, the Hadoop system follows a client-server-architecture which is reflected in our model: The environment of the SuT is formed by the client, that is the component which has the most influence on the SuT. Besides the client, the nodes and their connections to the *YARN* controller are also part of the environment, i.e., the controlled environment. This differentiation is of importance

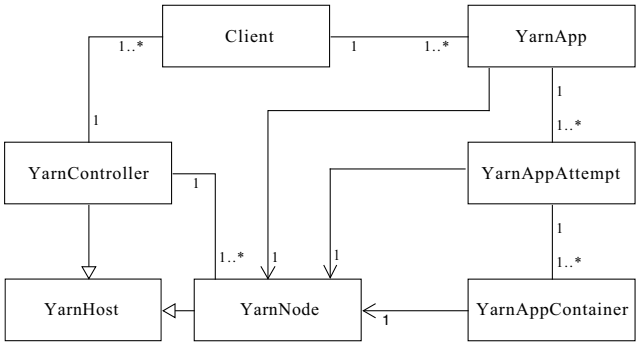

Fig. 2. Graphical representation of a simplified version of the domain test model formed by the classes describing the SuT as well as its environment in the S# test model. `YarnHost` represents the basic class for all distributed components in the cluster. The `YarnNode` executes a `YarnApp` allocated by the `YarnController`. The `YarnController` is the adaptive part of Hadoop. The `Client` is the environment which is not controlled by the SuT. The `YarnController` is responsible for allocating a client's task (formulated as `YarnApps` that have different `YarnAppAttempts` stored in the `YarnAppContrainer`) in the SuT.

```
/* ... */
AdjustmentNeededConstraints = new List<Func<bool>>
{
    () => YarnController.AvgResponseTime >
          Model.HighResponseTimeValue ||
        YarnController.AvgResponseTime <
              Model.LowResponseTimeValue  ||
        YarnController.TotalServerCosts > Model.MaxBudget *
              0.75
};
/* ... */
```

Listing 1. Partial S# component representing the constraint based oracle.

as the controlled environment is also used by the oracle, since parts of the functional requirements concern this control task. The other part of the environment, i.e., the client, is not controlled by the SuT, it is nevertheless interacting with the Hadoop system and driving the execution of the SuT, i.e., the tasks or requests sent by instances of the client class.

2) Oracle Constraint Model: The basic idea and the concept of the constraint based, automated oracle has been developed and described in depth in [7], [8]. Constraining invalid configurations within the adaptation loop enables to respect the characteristics of adaptive software systems, that demand for degrees of freedom to enable autonomous decision making. First, it is necessary to transform the requirements to constraints in order to check whether or not the requirements are fulfilled, as shown in [11]. A key advantage of defining these constraints on the domain test model is that it is possible to abstract from distribution. Thus, the constraint can be defined given a synchronized system. Indeed, the synchronization needs to be provided by connecting the model with the actual system, as described in section V. Listing 1 shows an excerpt of the constraint based oracle used for the Hadoop case. The shown constraint describes, in parts, the requirement that parameters of the configuration need to be updated and cause an adaptation. The constraint checks three rules that implies an adaptation of the system, i.e., a response time outside the specified slot and an exceedance of the budget. The constraints are formulated on the basis of the `YarnController` containing the necessary information.

B. Dynamic Test Model

The dynamic test model serves mostly as a definition of the test suite. However, not in the classical manner of representing sequences of test cases to be executed. This is due to the characteristics of the SuT the approach is designed for: the adaptive software systems. Since, a main motivation is to face adaptive systems with adaptive tests the concept of shifting decision from design time to run-time needs to be applied to our test setting as a fundamental concept of adaptation. This enables for adapting the test execution by the test automation during test execution, i.e., the run-time of testing as we consider it. Hence, the test suite is described by two kinds of models: one for the controlled part of the environment of the SuT and one for the dependent environment, i.e., the client. This distinction is also described in the static model. The first part of the test suite is based on a fault based test case description, the environment fault injection. The latter part describes the environment as a probabilistic test model, able to deliver endless test inputs, the environment profiles.

1) Environment Profiles: Environment profiles are probabilistic models, Markov models, that describe the interaction of the environment with the SuT. In case of Hadoop, the interaction is focused on the client, that is able to submit tasks and consequently controls the workload. Testing the *YARN* controller demands a workload on the Hadoop system in order to activate it. The basic idea of the environment profiles is to generate test inputs that represent the most likely conditions. It is up to the test engineer to design a good environment profile for the test suite. For forming the environment profile for the Hadoop case we used the most popular three benchmark collections that apply to Hadoop as well as information for empirical studies on the usage of Hadoop available in the literature [12]–[14]. These benchmarks are used for tweaking the parameters of *YARN*, among other things. The three benchmark collections are Hadoop MapReduce Examples, Intel's HiBench[3], and Statistical Workload Injector for MapReduce (SWIM)[4]. These have been clustered to extract the different possible tasks for Hadoop, resulting in 14 different types of actions that are grouped in four categories:

1) *Generators*
   - Text files: random text writer (rtw) and TestDFSIO -write (dfs-w)
   - Binary files: randomwriter (rw) and teragen (tgen)
2) *Data Processing*
   - Read: wordcount (wc) and TestDFSIO -read (dfs-r)
   - Sort: sort for text data and terasort (tsort) for binary data
   - Validate: testmapredsort (tstsort) and teravalidate (tval) for any sorting application
3) *Calculation*
   - pi: Quasi-Monte Carlo method for calculating π
   - pentomino (pent): solving the pentomino problem
4) *Simple Interaction* sleep and fail

After identifying the states, the transition probabilities have to be defined. In order to figure out these values we analyzed the benchmark as well as other common applications for Hadoop and the remarks of Zhang et al. [6] in detail. Further, we used the empirical analysis from the literature [12]–[14] to ground our numbers. The result is the transition matrix shown in table I with the transition probabilities used in the environment profile. The states shown are the 14 actions categorized above. Thus, a state change implies stopping one action (or completing it) and starting the next which is corresponding with the next state.

2) Environment Fault Injection: The second part of the test suite is formulated as environment faults. The faults are injected into the controlled environment of the *YARN* controller, i.e., the nodes and the connections between nodes, controller, and client. Any fault can be specified as `transient` or `persistent`, stating whether the environment fault is only active for one test step or for all remaining. Further, the environment faults are complemented by an activation criterion. This criterion enables to specify the intention of the abstract test case, e.g., only if few servers are active, enabling to use this information for adaptive test execution. In [5] we explained how a planner can be used to instantiate the abstract test cases, i.e., which case should be activated at which step, and in [4] we presented a rule-based reasoner, whereas the rules are a part of an apriori specified activation criterion. Both cases are applied in the Hadoop case in different runs.

Listing 2 shows the possible specification of an environment fault to be injected in S#. The component shown, in this simplified version of the `Node` class defines different properties of the node as well as functions. The functions are used to represent the functionality of the component of the SuT and also for mapping the test model with the actual SuT for test automation. Since the node is part of environment of the controller we implemented different test cases in form of environment faults. There is a transient and a persistent one; the persistent one is further using the rule based description for enriching the abstract tests. The first parameter is mapped to a boolean function used to describe the desired situation where the test case should be activated. The second parameter is set to `auto` in order to signal that the planner should select the number of nodes at run-time that should be injected with the fault if the described situation `TooFewServers` is present. No annotation, as for the transient fault, implies that the fault is activated at random.

V. S# TESTDRIVER FOR HADOOP

In order to fully automate the testing within S# it is necessary to connect the SuT, here the Hadoop system, with the executable S# model. The connection is established by a test driver which is integrated in the S# code, written in C#. To enable testing as described in section IV two functionalities must be provided by the test driver: (1) controlling the SuT by enabling the injection of faults (cf. section IV-B2) in the controlled environment of the SuT and (2) monitoring the SuT

[3]https://github.com/intel-hadoop/HiBench
[4]https://github.com/SWIMProjectUCB/SWIM

	dfw	*rtw*	*tg*	*dfr*	*wc*	*rw*	*so*	*tsr*	*pi*	*pt*	*tms*	*tvl*	*sl*	*fl*
dfw	0.600	0.073	0	0.145	0	0	0	0	0.073	0.073	0	0	0.018	0.018
rtw	0.036	0.600	0	0	0.145	0.036	0.109	0	0.036	0	0	0	0.019	0.019
tg	0	0.036	0.600	0	0	0	0	0.255	0	0.073	0	0	0.018	0.018
dfr	0	0.073	0	0.600	0	0.036	0	0	0.145	0.109	0	0	0.018	0.019
wc	0.073	0.109	0	0	0.600	0	0.073	0	0.073	0.036	0	0	0.018	0.018
rw	0	0.073	0.073	0	0	0.600	0	0	0.109	0.109	0	0	0.018	0.018
so	0	0.073	0.036	0	0.073	0.036	0.600	0	0.073	0	0.073	0	0.018	0.018
tsr	0	0	0	0	0	0	0	0.600	0.109	0.073	0	0.182	0.018	0.018
pi	0.145	0.109	0	0	0	0	0	0	0.600	0.109	0	0	0.018	0.019
pt	0.109	0.109	0	0	0	0.073	0	0	0.073	0.600	0	0	0.018	0.018
tms	0	0.145	0	0	0	0.073	0	0	0.036	0.109	0.600	0	0.018	0.019
tvl	0.073	0.109	0	0	0	0	0	0	0.109	0.073	0	0.600	0.018	0.018
sl	0.167	0.167	0.167	0	0	0.167	0	0	0.167	0.167	0	0	0	0
fl	0.167	0.167	0.167	0	0	0.167	0	0	0.167	0.167	0	0	0	0

TABLE I
TRANSITION MATRIX OF THE ENVIRONMENTAL PROFILE WITH THE PROBABILITIES USED IN THE TEST AUTOMATION OF THE HADOOP SYSTEM.

```
class YarnNode : Component {
  YarnController _connectedYarnController; bool _isActive;

  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 2. Simplified S# component representing a Hadoop `Node`.

with its controlled environment as well as the clients for the Hadoop system. Since the SuT and the test system is part of a distributed cluster, a connection between the test system and the SuT needs to be established. We use *SSH* to establish this connection. Thus, it is possible to use command line scripts to execute the control commands and to gather information from the Hadoop system for monitoring. Using command line scripts enables to use the full power of the interface supplied by Hadoop for the test driver. In order to keep the test system architecture unaffected from the concrete test driver implementation the test driver is encapsulated in a particular interface. This interface can be also used with an implementation of a *REST*-based test driver. The main functionality within the test driver implementation is to translate and transfer commands for controlling the SuT and to receive and translate monitoring information. The counterpart in the Hadoop system which is needed are the scripts used to supply the relevant functionality for controlling and monitoring.

A. Controlling the SuT

The SuT is controlled by the test driver on the one hand by injecting faults into the controlled environment of the SuT, i.e., the activation of environment faults, and on the other hand by sending a workload to the Hadoop system. The later one can be directly generated at the test system and needs no tunneling through the *SSH* connection. The workload is generated by having function calls to Hadoop for the 14 different classes of actions (cf. section IV-B1 and table I). The functions make use of the workloads supplied by standard benchmarks we used for extracting the states of our environment profile. They are called from C# and thus directly executed from the test framework. Fault injection instead needs an *SSH* connection to the SuT host(s). We generated command line scripts for executing faults, making use of the supplied functionality of Docker which is hosting the SuT. Thus, it is for example possible to disable a network connection or to disable/shutdown a particular node of the Hadoop system as a fault activation.

B. Monitoring the SuT

Monitoring is needed in order to update the run-time model of S# after every step (cf. section II-A). This is the foundation for enabling the adaptive test automation. The execution order of the steps is fixed and determined by the test engineer. In our case, we first updated the state of the model and afterward let the planner execute the selected test steps after the workload is sent to the SuT as calculated by the environment profile. Executing the steps is in the responsibility of the test driver as described before. For updating the model we need to extract that information for the system as a snap shot. Indeed, in a distributed system the generation of a consistent snap shot is far from obvious. However, as elaborated in [4] small time difference in states of different system parts do not have great impact on the overall result. Thus, we gather the information at every test step (a test step lasts for a maximum of 300ms) by sampling the information in a fixed order. The resulting time difference has no impact on the overall results, as explained in section II-B. The test domain model (cf. fig. 2) is instantiated as a run-time model. Thus, the information for its attributes has to be retrieved from the SuT, that is available by on the one hand the Hadoop system itself (making use of the *graphite* extension, cf. fig. 1) and on the other hand the Docker ecosystem. The data is retrieved by command line functions and needs to be extracted by a parser afterward. This parser is written in C# and maps the information into the S# model.

After the test driver is defined once the test engineer is able to abstract from this technical details and from synchronization by defining tests to be automated or the constraint based oracle only on the consolidated model.

VI. EXPERIENCES

The experiences made by applying [4], [5] in the Hadoop case study is summarized in the following, reflecting the

abilities of our approach on testing adaptive system within an adaptive test automation.

a) Model-based, adaptive test automation of an adaptive Hadoop System application: In [4], [5] we explained how awareness in a model-based test automation is able to give the test the ability to act adaptive. Work by Zhang et al. [6] showed that adaptive systems are not limited to artificial research case studies, they can be applied to a real world application. In this paper, we showed that our concepts for model-based, adaptive test automation are also applicable for this particular real world application: the self-adaptive controller of a Hadoop application. The central concept of our approach is to use run-time models; the underlying model-based paradigm enables to handle complex systems—here a distributed Hadoop system—by abstraction. Abstraction makes it easy to integrate the automated oracle, without worrying about distributed aspects, and further enables to enrich and define the test suite in a way that it can be used in an adaptive test automation. The key concept for adaptive test automation is on the one hand to give the adaptation mechanism the needed freedom to act autonomously and on the other hand to add the information about the intention of the test case in order to act in an intelligent way. The latter one is done by descriptions based on the model state. Thus, the test model used is an enabler for adaptive test automation.

b) Implementing test harness for distributed test environment needed for an adaptive test automation: Indeed, the model-based paradigm made things easier by abstraction. However, test automation still needs somehow to cope with the complexity of the system when tests are executed and evaluated. This is done by the test driver we integrated into our test system. The model defined which kind of information is needed to be extracted from the SuT and which information or actions needed to be executed on the SuT. The set of command line functions we defined under the hood of the test automation is still not as generic and as reusable as we would like it to have. It needs to be customized for each and every application by a test engineer. We showed that and how it is possible to do so for a complex real world application. In future work, we focus on the challenge of connecting a complex system to our test ecosystem in a generic (maybe learning) way.

c) Run-time reflection of a distributed Hadoop System in our S# environment: Having coped with the aforementioned challenge it turns out that the task of the test engineer is much more focused on its actual task: defining and designing test cases, the automated oracle, and selecting the right degree of abstraction. The run-time reflection is the enabler for a more focused task of the test engineer, but further allows to specify tests with more information: the intention of the test case in a run-time context. That enables the adaptation of the test automation at run-time. Making test runs possible, that could not be specified without this information. This leads to a quite small test suite defined by the test engineer, that results in a quite large number of actually executed test cases at run-time.

d) Case study for testing a distributed, adaptive real world software systems: We showed that the concept shown in [4], [5] have been successfully applied to a distributed, adaptive real world software system. The model-based concept made it possible to handle the complex, distributed system. Nevertheless, the task of technically connecting the SuT with the test system is still highly customized and challenging. In total we executed over $300,000$ test cases on our Hadoop instance. But, we failed to reveal any failure during testing, indeed, this is showing the stability of a widely-used established commercial software. Simplistic errors have been injected as mutants in the system, these have been revealed as failures during testing. However, we claim that there is a need for more complex mutants, that are especially tailored to adaptation mechanisms or in general autonomous systems in order to evaluate the mutation score in depth of our approach. This will be part of our future work.

ACKNOWLEDGMENT

This research is sponsored by the research project *Testing self-organizing, adaptive Systems (TeSOS)* of the *German Research Foundation*.

REFERENCES

[1] M. Salehie and L. Tahvildari, "Self-adaptive Software: Landscape and Research Challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, 2009.

[2] R. de Lemos *et al.*, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap," in *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.

[3] R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds., *Software Engineering for Self-Adaptive Systems III. Assurances*, ser. LNCS, vol. 9640. Springer, 2017.

[4] B. Eberhardinger, A. Habermaier, and W. Reif, "Toward Adaptive, Self-Aware Test Automation," in *12th Int. Wsh. Automation of Software Testing, (AST2017)*. IEEE Comp. Soc. 2017, pp. 34–37.

[5] B. Eberhardinger, H. Seebach, A. Reichstaller, A. Knapp, and W. Reif, "Adaptive Tests for Adaptive Systems: The Need for New Concepts in Testing for Future Software Systems," *Softwaretechnik-Trends*, vol. 38, no. 1, 2018.

[6] B. Zhang, F. Křikava, R. Rouvoy, and L. Seinturier, "Self-Balancing Job Parallelism and Throughput in Hadoop," in *Distributed Applications and Interoperable Systems*, ser. LNCS, vol. 9687. Springer, 2016, pp. 129–143.

[7] B. Eberhardinger, G. Anders, H. Seebach, F. Siefert, A. Knapp, and W. Reif, "An Approach for Isolated Testing of Self-Organization Algorithms," in *Software Engineering for Self-Adaptive Systems III. Assurances*, ser. LNCS, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, Eds., vol. 9640. Springer, 2017.

[8] B. Eberhardinger, A. Habermaier, H. Seebach, and W. Reif, "Back-to-Back Testing of Self-organization Mechanisms," in *28th Int. Conf. Testing Software and Systems (ICTSS 2016)*, 2016, pp. 18–35.

[9] A. Computing *et al.*, "An architectural blueprint for autonomic computing," *IBM White Paper*, vol. 31, pp. 1–6, 2006.

[10] Spirals-Team. (2018, Mar.) Github: Spirals-team hadoop. [Online]. Available: https://github.com/Spirals-Team/hadoop-benchmark

[11] B. Eberhardinger, J. Steghöfer, F. Nafz, and W. Reif, "Model-driven Synthesis of Monitoring Infrastructure for Reliable Adaptive Multi-Agent Systems," in *24th Int. Symposium on Software Reliability Engineering, (ISSRE 2013)*, 2013, pp. 21–30.

[12] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, 2012.

[13] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's Adolescence: An Analysis of Hadoop Usage in Scientific Workloads," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 853–864, 2013.

[14] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.