Int J Softw Tools Technol Transfer (2017) 19:271–279
DOI 10.1007/s10009-017-0449-2

INTRODUCTION

Model-based testing as a service

Steffen Herbold[1] · **Andreas Hoffmann**[2]

Published online: 6 March 2017
© The Author(s) 2017. This article is published with open access at Springerlink.com

Abstract The quality of Web services is an important factor for businesses that advertise or sell their services in the Internet. Failures can directly lead to fewer costumers or security problems. However, the testing of complex Web services that are organized in service-oriented architectures is a difficult and complex problem. Model-based testing (MBT) is one solution to deal with the complexity of the testing. With MBT, testers do not define the tests directly, but rather specify the structure and behavior of the System Under Test using models. Then, a test strategy is used to derive test cases automatically from the models. However, MBT yields a large amount of tests for complex systems which require lots of resources for their execution, thereby limiting its potential. Within this article, we discuss how cloud computing can be used to provide the required resources for scaling up test campaigns with large amounts of test cases derived using MBT.

Keywords Testing as a service · Model-based testing · Cloud computing · TTCN-3

1 Introduction

In the last decades, software started to control many aspects of our everyday lives. Bad software quality can lead to all kinds of failures:

✉ Steffen Herbold
herbold@cs.uni-goettingen.de

Andreas Hoffmann
andreas.hoffmann@fokus.fraunhofer.de

1 Institute of Computer Science, University of Goettingen, Goettingen, Germany

2 Fraunhofer Institute FOKUS, Berlin, Germany

– In August 2015, a software failure at the bank HSBC delayed 275,000 payments, preventing thousands of people to get their pay checks in time.
– At the beginning of 2016, HSBC was hit by another software failure, preventing millions of costumers to access their online accounts for two days.
– The software of a smart thermostat failed after a broken update in January 2016 drained the energy of the thermostat rapidly, due to which users were unable to use their heating in the midst of winter.

These are just some recent examples of a very long list of high-profile and high-impact software failures.

As part of the growing success of the Internet, Web services like online banking became an important means to offer services to costumers. This gave rise to service-oriented architectures (SOAs), a paradigm where the software is decomposed into single Web services that are coupled loosely with each other [9]. An important aspect of SOAs is that the Web services are only described using their interfaces, independent of their implementation of location. SOA applications combine Web services and define patterns for their interactions. This process is called service orchestration. A good example for SOA applications is modern travel booking portals on the Internet: airlines, hotels, and travel agencies offer Web services through which information regarding their offers can be accessed and booking orders can be placed. Aggregating Web sites build a SOA application by orchestrating these services and offering, e.g., to search flights by different airlines at once and compare their prices.

A high quality is critical for the success of Web services. If you consider the above example, a failing Web service of an airline means that the flights of the airline would not appear in booking portals anymore, which would directly lead to fewer costumers. Even if the service does not fail directly,

a security issue in the booking system could be exploited to place invalid orders. However, testing of single Web services in isolation is not sufficient as one must make sure that the Web service can function within a full SOA application and not just on its own. Thus, the known orchestrations should be tested as well. This leads to several challenges for testing service-centric solutions [5,6], for example:

- The usage of services can rapidly change if they are used in new or different orchestrations.
- The source code or other structural information of services is often not available, only their interfaces. This prevents the usage of white box testing techniques. Moreover, this complicates the definition of correctly mock objects required for service unit testing.
- The unanticipated evolution of services by other providers within a service orchestration can lead to all kinds of complications for the operation and quality assurance of services, e.g., due to changing interfaces or behavior.
- Setting up a test environment with all required components like application servers, firewall configurations, and monitoring is itself already a challenging task.

Model-based testing (MBT) provides a solution for many challenges for software testing. Models provide a high level of abstraction that allows to define complex behavior in a compact way. Due to this, models like state machines can capture the behavior of whole protocols and orchestrations and are, therefore, suited for SOA testing. The number of tests that is derived from a model depends on the test strategy and the complexity of the software that is modeled. Typically, the test strategies try to gain a certain coverage of a system, e.g., by executing all events of a system or by putting the system in all its logical states at least once. However, more complex coverages, e.g., where all pairs or even triplets of events that are possible should be covered, can usually not be achieved due to the exponentially growing number of tests involved. This is especially problematic for large and complex software like orchestrated services with many possible interaction paths. Thus, the testing efforts are limited by the resources available for the test execution.

To overcome the resource limitations, cloud computing is a viable tool. With cloud computing, it is possible to rent computing infrastructures on demand. Moreover, one feature of cloud computing is elasticity, which allows the dynamic scaling of computing infrastructures based on the current computational needs. This is a natural fit for scaling MBT and enables large-scale test campaigns with automatically generated tests for complex SOA orchestrations.

Within this article, we introduce the general concepts of MBT using an example of a simple Web service as System Under Test (SUT) and Unified Modeling Language (UML) for modeling in Sect. 2. Then, we recap the principles of cloud computing in Sect. 3 including the characteristics of clouds that impact the MBT and the service models of cloud providers. Once these foundations are established, we discuss how the cloud can be used to define a platform for the development of MBT solutions and how testers can use such a platform in Sect. 4. This discussion is based on the results of the MIDAS European project [12,18]. Finally, we conclude the article in Sect. 5.

2 Model-based testing

The International Software Testing Qualifications Board (ISTQB) defines MBT as "testing based on or involving models" [15]. This means that a conceptual model of the SUT is used to derive tests for the system. This definition is rather generic and allows for different test artifacts that can be derived from models, for example:

- Abstract test cases that provide high-level task descriptions for manual testing. The test cases may not contain all values required or may be missing obvious intermediate steps, which can lead to variations when concrete test cases are defined based on the abstract tests.
- Concrete test cases that provide all required information for repeatable manual testing. It may be possible to automate the generated tests with manual effort.
- Concrete test cases that are available as an automatically executable test script which can be compiled and executed.

Our aim is to outline how MBT can be used to generate a massive amount of test cases for automated and scalable test executions on the cloud. This is only possible if the test execution is possible without any manual interaction, since manual interaction would directly negate the aim of running massive amounts of tests. Therefore, we only consider the generation of automatically executable test scripts from models and all future usages of MBT only refer to this fully automatable scenario.

For automated MBT, it is important that the test model is rich in terms of detailed information about the SUT; specifically, three things are required: a behavioral model, interface descriptions, and deployment information. In the following, we will use UML to give examples how each of these information is modeled. Throughout this, we will use a running example with two Web services to explain the concepts: one that sells products and another that supplies the materials. The example is a simplified version of a logistics prototype. Details on the complete example are discussed within this special section of the articles by Herbold et al. [13] and Barcelona et al. [3].

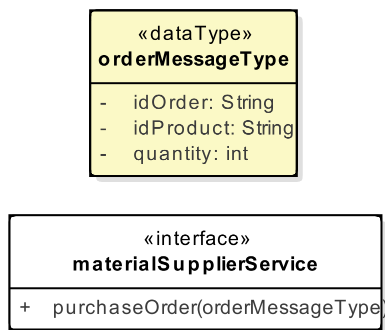

Fig. 1 An example for a UML class diagram

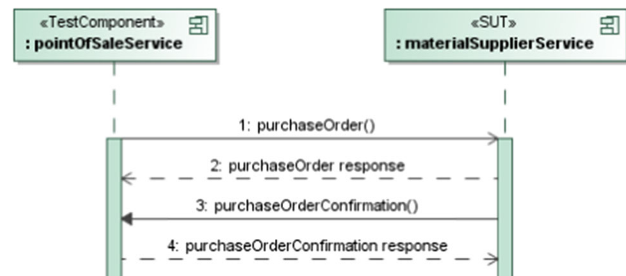

Fig. 2 An example for a UML sequence diagram

2.1 Structural description

The first requirement for automated MBT is a structural description of the SUT comprising the interfaces exposed by the SUT. The interface description must describe the operations that may be called as well as the data structures used for input and output by these operations.

UML class diagrams provide a convenient way for both. UML class diagrams are basically rectangles that are separated into compartments. Each rectangle represents a type, and the compartments contain the operations and data associated with the types. To define the interface of the SUT, the model must contain both operations and data types. Figure 1 shows an example that defines the interfaces and data types for our material supplier service. The service exposes one operation with the name `purchaseOrder`. This operation requires one parameter, which is of type `orderMessageType`. This message type is also defined in the diagram and contains two strings and one integer value.

2.2 Behavioral models

The second requirement for automated MBT is a behavioral model of the SUT. The task of the behavioral model is to define how the SUT should behave when it is interacting with its environment. Based on this definition, a test strategy is used to derive test cases from the behavioral model. The combination of behavioral model and test strategy decides which and how many test cases are derived from a model. In the following, we give two common examples. First, we show how UML sequence diagrams can be used to define a single test case with a very simple test strategy. Second, we show how UML state machines and a complex test strategy can be used to derive thousands of test cases.

At the top of a sequence diagram, objects are defined. These objects represent the components of the SUT and the test environment. Each object has a lifeline. The lifelines define at which point in time an object exists. The rectangles on the lifelines define when objects are active. The communication between objects is defined using messages defined by arrows. The messages are associated with operations that are defined on the interface of the underlying component (see Sect. 2.1). The behavior of the objects is defined by exchanging synchronous and asynchronous messages between objects. UML sequence diagrams also allow more complex concepts like conditional execution and loops, but we will skip these in this brief introduction.

Figure 2 shows an example for a UML sequence diagram. The objects in the diagram represent our two services: `pointOfSaleService` and `materialSupplierService`. The diagram defines a valid sequence of communication between the two services. First a `purchaseOrder` message is sent to `materialSupplierService` upon which it responds and subsequently sends back a `purchaseOrderConfirmation` to the `pointOfSaleService`, which also responds. To allow the definition of a test strategy, the two objects are annotated: the `pointOfSaleService` is a test component, the `materialSupplierService` the SUT. A simple test strategy is then to take the depicted behavior as is and transform it into a test case for the testing of the `materialSupplierService`: the behavior of the `pointOfSaleService` is simulated by the test environment which also checks whether the `materialSupplierService` answers correctly.

A second way to define behavioral models we demonstrate is UML state machines. Using a state machine, one can define how a system reacts on events depending on the state it is currently in. The UML state machines themselves consist of a set of states, depicted by rectangles with rounded edges, and transitions between these states, depicted by arrows. The transitions can be due to events, fulfillment of conditions, or even automatic. State machines can have additional features like an initial state, end states, nested states, and parallel executions, which we skip for this brief introduction.

Figure 3 shows a small example of a very simple state machine that could be the underlying behavioral model of the material supplier service. Since we have no start or end states, we implicitly assume that the service is running forever. The service has three states, i.e., it behaves differently if it is waiting for an order, has received an order, or has con-

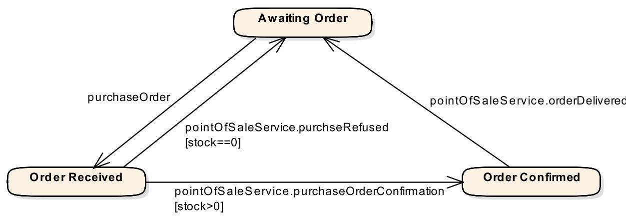

Fig. 3 An example for a UML state machine

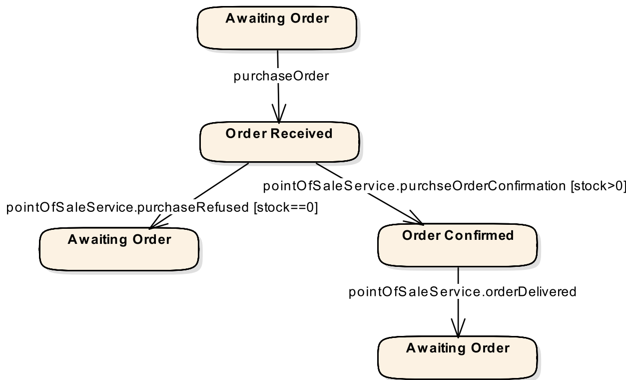

Fig. 4 Transition tree derived as round-trip paths from the UML state machine depicted in Fig. 3. Each path from the root node to a leaf node is a test case

firmed an order. In the state `Awaiting Order`, the service only reacts to `purchaseOrder` calls. Once such a call is received, it switches the state to `Order Received`. If the product is not in stock, the point of sale is notified and the state of the material supplier goes back to `Awaiting Order`. If the product is in stock, the order is confirmed to the point of sale, and the state of the material supplier switches to `Order Confirmed`. Once the order is delivered, the material supplier notifies the point of sale service and the state is changed to `Awaiting Order` again.

Many strategies were suggested for the generation of test cases from state machines. Within this article, we use the coverage of round-trip paths [4]. The round-trip paths are determined using the transition tree of the state machine. The root of the transition tree is the initial state of the state machine. Since we have no explicit starting state, we use the awaiting order state instead. Starting from the root, one adds all outgoing transitions from the state to the tree. This is then repeated recursively for the newly created tree nodes. The recursion ends if a state was already visited on a higher level of the tree. Figure 4 shows the transition tree for the state machine depicted in Fig. 3. The paths both end in the state awaiting order, because it was already visited on a higher level of the tree. From the transition tree, one can directly derive the sequences for test cases as all paths from the root node to a leaf of the tree. Hence, the transition tree in Fig. 4 defines two test cases. With larger automata for more complex services, the size of the transition trees grows

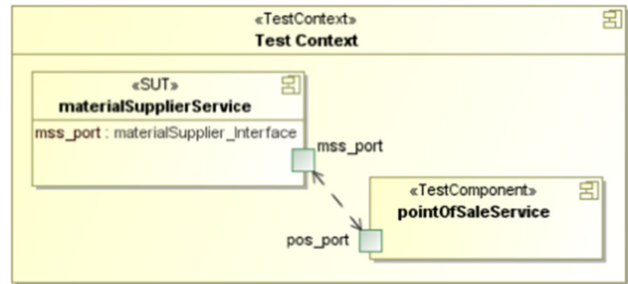

Fig. 5 An example for UML component diagram that defines a SUT deployment

exponentially and a very large amount of test cases can be generated.

2.3 Deployment information

The third requirement for automated MBT is the deployment information of the SUT. There are two locations where this information could be contained.

1. The information is being part of the test harness where the test cases are executed. In this case, this information is not contained in the model, but added directly during the test execution.
2. The information is being contained in the model and part of the generated test cases.

To exemplify how both work, we use a hybrid example here. The structure of the SUT deployment is defined using a UML component model. With a UML component model, one can depict components that can be connected using ports. These ports realize interfaces, i.e., they offer the operations of the associated types. Moreover, components can contain other components.

In Fig. 5, we show an example for a UML deployment diagram. The diagram shows three components: the `Test Context` which contains the `materialSupplier Service` and the `pointOfSaleService` components. Same as in the example of the sequence diagram (see Fig. 2), the `materialSupplierService` is marked as the SUT and the `pointOfSaleService` as the test component. Both components offer one port. These ports offer the interface defined by the structural model of the SUT (see Fig. 1). A connection between these ports defines that the components can communicate via these ports.

Thus, the structure of the SUT deployment within the test context is defined. However, to automate test cases, additional information is required: the endpoints of the services, both of the material supplier service in its role as SUT, and the point of sale service which is used as test component within the test context. This information is not contained in

the model. Instead, it needs to be defined prior to execution within the test harness of the test execution. This is usually done by a configuration file or program argument and depends on the software used for execution.

3 Cloud computing

Within this section, we want to introduce the basic concepts of cloud computing. The term is used quite frequently with different meanings depending on the context, applications, and audience. Within this article, we follow the terminology introduced by the National Institute of Standards and Technology (NIST) of the US Department of Commerce.

Definition 1 (*Cloud Computing*) A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [16].

This definition is widely accepted within the cloud community. In laymen's terms, it means that clouds offer convenient and flexible access to resources that are hosted elsewhere, without the requirement to know how exactly resources are hosted. Together with this definition, the NIST also defines which characteristics cloud resources should fulfill and which service models are offered to consumers of cloud services.

3.1 Characteristics

The NIST definitions specify five essential characteristics of cloud computing [16]. The two most important for the usage of cloud infrastructures for testing are *rapid elasticity* and *on-demand self-service*.

The availability of on-demand self-service means that no human interaction is required in order to gain access to the capabilities of a cloud service. This means that a consumer can request any resources unilaterally, without, e.g., contacting a sales person first or negotiating pricing. All services are offered transparently and are—oversimplified—"just one click away." This does not only entail the buying of services and their provisioning, but also the releasing of services. This leads to a pay-per-use business model, where you only need to pay for the resources you are actually using. Once the usage is completed, the resources are released and no further payment is required.

Rapid elasticity means that, depending on the required computing capabilities, the cloud services provider can scale up (or down) the required resources dynamically. In theory, these scalability features are unlimited and can be used at any time. A good example for elasticity is load balancers for

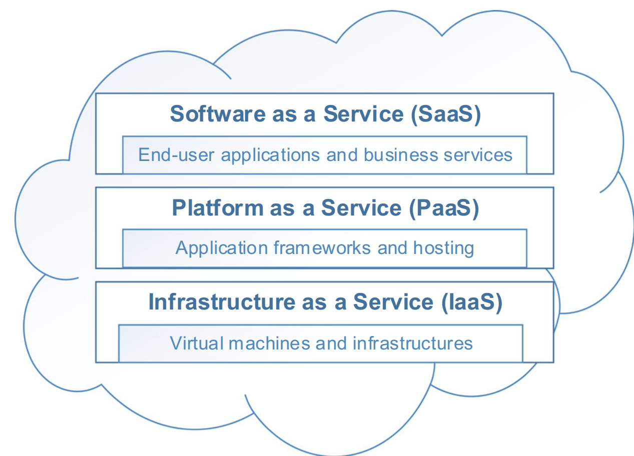

Fig. 6 Cloud computing service models

Web sites. In case of heavy load, e.g., during lunch hours at a news Web site, new machines for handling the incoming traffic can be added and used via a load balancer. Once the number of users drops again, these additional resources can be released.

3.2 Service models

Cloud services are principally divided into three service models that define which capabilities are provided to consumers of cloud services by a cloud provider [16]: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) (see Fig. 6). The service levels define a rough categorization based on how much influence the service consumer has on the applications running on the cloud. Common for all service levels is that the consumer has no direct influence on the cloud's hardware infrastructure itself.

The most influence is granted to the consumer in the IaaS model. Here, the consumers are very flexible and are offered a broad array of low-level access, including the management of operating systems, the software installed, and direct access to the system storage. For the most part, this means that it is possible to deploy arbitrary (VMs) and manage their content. IaaS may also offer additional capabilities like the definition of (virtual) networks between deployed VMs and sometimes even the definition of firewall rules.

With PaaS the consumer still has control over the software running on the cloud infrastructure. However, in contrast to IaaS no direct access to the operating system is given. Instead, the cloud provider supports certain services, e.g., deployment of Web services on an application server or the usage of predefined storage system like an object storage. Thus, PaaS users do not have to care for the specifics of operating systems and their configurations, but also cannot deploy arbitrary software but only what is supported by the cloud provider.

With SaaS, consumers can access fully running applications. With SaaS the consumers have no control about the applications logic anymore and cannot deploy their own software. Instead, they use a pre-configured and installed software they simply access with a client, which is often just the Web browser.

4 Testing as a service with MIDAS

Now that we introduced the concepts behind MBT and cloud computing work, we come back to our initial vision from the introduction: massively scaling up test campaigns using MBT and cloud computing. The general concept behind this idea is quite simple: generate a huge amount of tests using MBT and use the elasticity of the cloud to scale the test execution. However, there are several conceptual and technical problems that must be resolved to achieve this. Within this section, we will outline the problems and how to overcome them. We will discuss how an IaaS cloud can be used to define the infrastructure for a PaaS solution for testing. By implementing applications on this platform, we show how a MBT test application can be deployed on a cloud to create a Testing as a Service (TaaS) solution.

This discussion will mainly be based on the results of the MIDAS European project [18]. The aim of MIDAS was the development of such a platform for the testing of Web services based on the SOA, specifically the orchestration of multiple services. The problem that exists here is that services by themselves can already be quite complex and offer lots of operations. If multiple services are used together in a so-called service orchestration, this complexity grows exponentially. Hence, the testing of such orchestrations is a hard problem. The approach of MIDAS was to use the advantages of MBT and cloud to solve this problem by allowing large-scale test campaigns.

4.1 Setting up the infrastructure

The first step to achieve TaaS is to create an appropriate cloud computing infrastructure. The approach chosen within the MIDAS project was to create a PaaS solution based on an existing IaaS infrastructure. The rationale behind this was that elasticity and pay per use can be achieved with IaaS. Moreover, IaaS allows for a nearly arbitrary definition of the software stack that is installed. While existing PaaS solution also supports elasticity and pay per use, they lack the required flexibility to allow for testing Web services, e.g., they do not provide a test execution engine and do not allow for potentially required firewall configurations in order to communicate with the Web services that shall be tested.

The PaaS of "Model and Inference Driven - Automated testing of Services architectures" (MIDAS) is based on the SOA paradigm, a quite common solution for PaaS services, e.g., supported by Amazon Web Services (AWS) [1] and by Google's AppEngine [11]. MIDAS provides SOAP interfaces based on Web Service Description Language (WSDL) that can be used for the definition of services for the MIDAS PaaS. The platform itself already provides a set of services which are required independent of testing:

- user authentication,
- object storage [17]-based file management, and
- accounting and billing.

Additionally, some services to support the test activities are provided:

- definition of test campaigns,
- test script compilation, and
- test execution.

The test scripts and execution are based on Testing and Test Control Notation version 3 (TTCN-3). All additional test services (i.e., the services that generate the TTCN-3 test cases from models) that are developed for MIDAS must implement the SOAP interfaces prescribed by the MIDAS PaaS (Sect. 4.2).

While the above sounds straight forward to implement, there is a major problem for many applications: many existing model-based tools as well as high-quality test execution engines are proprietary and require licensing. These licenses are usually bound to a single machine. However, for scaling an IaaS multiple VMs are created, thus requiring multiple licenses. Hence, while the provisioning of VMs with the required software installed may be unproblematic, the actual execution of the installed software may not be possible due to missing licenses. While this is a common problem for porting applications to the cloud, no general solution is yet available, because the underlying problem is not of technical nature, but rather due to the licensing models.

Within the MIDAS project, we faced these problems with our TTCN-3 compiler and execution engine. Both were based on TTworkbench [23]. Fortunately, Testing Technologies [22], the provider of TTworkbench, was so kind as to allow us to use a licensing server that could distribute licenses to VMs dynamically. Thus, we could scale up and down the usage of licenses with the currently required VMs for executing tests. We think that this approach provides the blueprint for a feasible solution to structure flexible licensing that allows the support of a pay-per-use model as is common for cloud applications. By monitoring which licenses are used for which amount of time, a payment structure between the tool provider and the cloud platform provider can be defined. More details on this problem, as well as other technical prob-

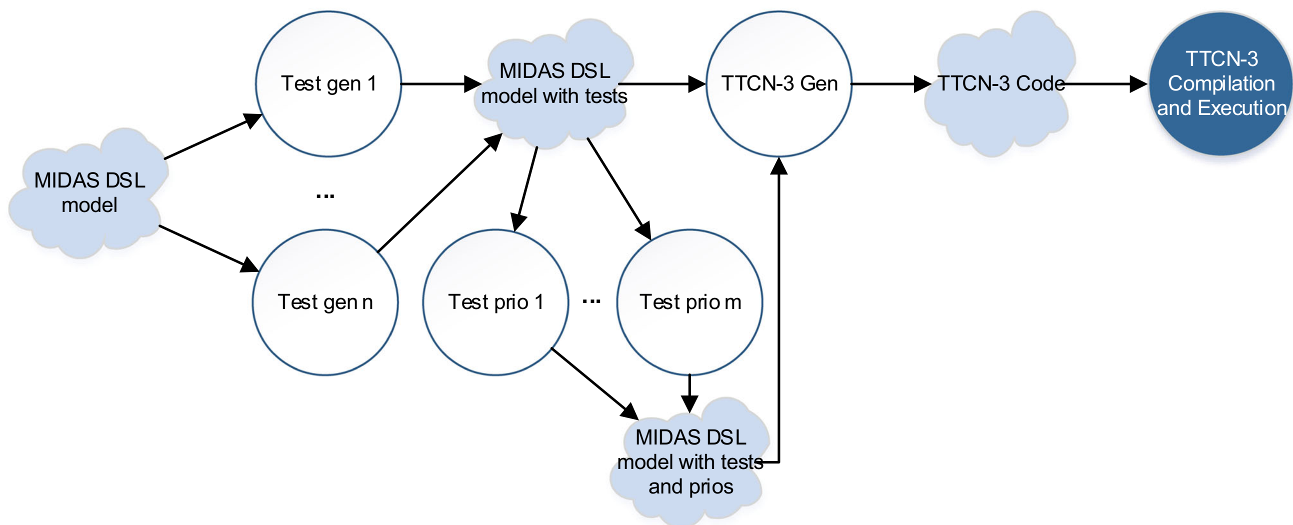

Fig. 7 A data flow diagram showing an example for test services and the information flow between them

lems which were solved within the MIDAS project, can be found in the literature [7,8].

4.2 Model-based testing on the cloud

Using the MIDAS platform, it is possible to plug in Web services that provide the capabilities required for MBT to the platform. In order to maximize the flexibility and scalability of the platform, each service should be defined as a closed unit, i.e., without directly calling any other service of the MIDAS platform, except for the services offered by the MIDAS platform for file management and user authentication.

The test services communicate via files. The idea behind this approach is to allow test services to exchange information as serialized representations of the artifact that was generated. Thus, if we have a test case generation service that uses the serialization format X, all other test services provided within MIDAS that can read X as input may use the result, e.g., test prioritization services, or TTCN-3 generation services. To demonstrate the power of this concept, MIDAS itself uses a modeling approach based on UML [20] augmented with concepts from the UML Testing Profile (UTP) [2] and some additional restrictions and stereotypes required for testing of SOA applications. The serializations of these models are the files that are exchanged between services and allow a seamless interaction between different services.

Figure 7 shows an example for services deployed on the platform and their interactions. Test case generation services may use a MIDAS DSL model as input to generate tests using different test strategies. The results from the test case generation can then be either directly used by the TTCN-3 generation service or first used by test prioritization services that define an order for the tests for their execution. Finally, the generated TTCN-3 is used by the platform components for compilation and execution against the SUT.

Since each of the services is defined as a single and isolated unit, new services can be added without much effort, e.g., a new test case generation service can be plugged in that follows a different test strategy. This makes the parts interchangeable and allows for the definition of a variety of test approaches. The only limitation, from a developers point of view, is the need to adhere to a commonly shared input and output format of the models that are exchanged between the files, i.e., in case of MIDAS the MIDAS DSL. As long as this format is used, one can reuse all existing services that also utilize the same approach.

However, even this limitation can be circumvented. Any exchange format is possible, if all services are replaced, i.e., also a TTCN-3 generation service is provided. This allows the usage of other MBT approaches not based on the MIDAS DSL. This is exemplified by the MBT approach based on Service Component Architecture (SCA) XML [19] and State Chart XML (SCXML) [25] discussed by Hillah et al. [14]. Thus, the architecture of the PaaS serves as a versatile way to define MBT approaches on a cloud infrastructure. However, the effort for developers is higher as a new implementation of all services is required, including test case generation and TTCN-3 generation.

4.3 Definition of test campaigns

The previous sections explained how a platform for testing can be built on the cloud and how MBT solutions can be implemented from a developers point of view. Once these services are available, the platform looks from a testers point

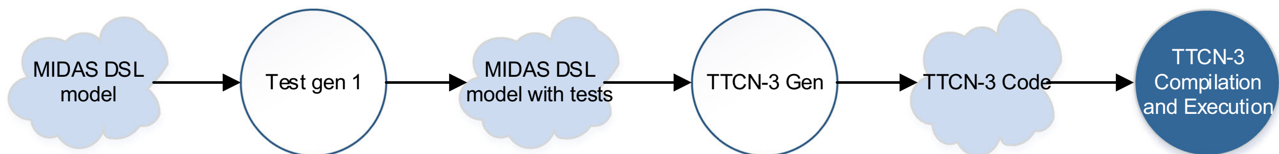

Fig. 8 An example for a concrete test campaign

of view like a SaaS application, to which we refer to as TaaS: all software is available and running, and the testers only need to provide the required inputs for running test campaigns.

These inputs are twofold: (1) a MIDAS DSL model and (2) an orchestration for the services provided by MIDAS. The DSL model provides the structural and deployment information about the SUT. Additionally, the DSL model must provide a behavioral model that can be used by the test strategies that shall be used by the test case generation services. The orchestration defines which of the services provided by MIDAS are called with which files as input and in which order. Basically, this can be thought of as one path through the data flow diagram of the service deployment we described above (see Fig. 7). Figure 8 shows an example of such a path. With this orchestration, the test generation service 1 is called; then, the TTCN-3 code is generated for the generated tests and afterward compiled and executed. The orchestrations are soft-coded. Thus, theoretically any order of services installed on the TaaS can be called. This is only restricted by the required inputs and outputs of the services which must be compatible.

5 Conclusion

Within this article, we discussed how MBT as a potential solution to deal with the complexity of the Web service orchestration testing can be scaled up using cloud infrastructures. We outlined how MBT can generate a massive amount of tests to a large for execution on normal commodity test hardware. Then, we explained how cloud computing's flexible elasticity and scaling mechanism together with its pay-per-use service model provides a solution that can be used to execute such a large amount of tests. However, a testing solution requires a specialized cloud platform, as certain needs like licenses for test software are not within the portfolio of test providers. Within the MIDAS project a service solution for software testing was developed and explored.

We believe that such a cloud-based testing platform will be a major part of how tests will be executed automatically in the future. The MIDAS project was not alone in considering this idea. In parallel, the Test@Cloud project [21] explored how only the execution of tests could be moved to the cloud, while all other test activities would still be performed locally. Another example for cloud-based testing that recently evolved is Travis CI [24], a cloud service for test execution for projects hosted on GitHub [10]. This trend shows that not only applications are moving to the cloud, but that quality assurance has started to follow this trend.

Articles selected for this special section

We selected three articles [3,13,14] for this special section. The articles provide insights into how MBT techniques can be moved to the cloud and how the challenges, e.g., due to licensing, firewalls, can be resolved. The articles provide different perspectives and are all the results of cooperations between industry and academia.

- The article "Automated and intelligent scheduling of distributed system functional testing" by Hillah et al. [14] discusses a testing solution based on model checking and intelligent scheduling for testing. The authors describe the theory behind using model checking in combination with machine learning to derive which test cases should be executed next in order to prioritize tests which are most likely to fail, due to already executed test cases. They demonstrate their approach using a cloud platform that schedules tests on the fly during the test execution dynamically. This paper addresses how the test execution on a cloud can be steered from another cloud component for dynamic scheduling.
- The article "Combining usage-based and model-based testing for service-oriented architectures in the industrial practice" by Herbold et al. [13] investigates how usage-based testing can be combined with traditional MBT and moved to the cloud. The approach includes monitoring of Web services, usage profile inference, test case generation, as well as generation of executable test cases and their execution. Through this, the authors evaluate which problems must be resolved when this workflow shall be implemented in a fully automated way on a cloud platform. Through their analysis, the authors provide solutions for the arising problems.
- The article "Practical experiences in the usage of MIDAS in the logistics domain" by Barcelona et al. [3] evaluates the MIDAS platform for MBT on the cloud from

an industrial perspective. The authors evaluate the difficulty of the modeling, the usability of the tooling, as well as the fault-finding capabilities of considered testing techniques. This way, the authors estimate the potential Return On Investment (ROI) of using MBT on the cloud. The estimation includes how cloud licensing models can be part of reducing costs.

Acknowledgements The editors of the special section wish to express their gratitude to all authors, reviewers, and the STTT editorial team for their contributions, help, and patience during the composition of this special section. The work on the special section was done in context of the MIDAS European project (Project No: 318786).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Amazon web services. https://aws.amazon.com/
2. Baker, P., Dai, Z.R., Grabowski, J., Haugen, O., Schieferdecker, I., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. Springer, Secaucus (2007)
3. Barcelona Liédana, M.A., López-Nicolás, G., García-Borgoñón, L.: Practical experiences in the usage of midas in the logistics domain. J. Softw. Tools Technol. Transf. (STTT) (2016)
4. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co. Inc., Boston (1999)
5. Canfora, G., Di Penta, M.: Testing services and service-centric systems: challenges and opportunities. IT Prof. **8**(2), 10–17 (2006). doi:10.1109/MITP.2006.51
6. Canfora, G., Di Penta, M.: Service-Oriented Architectures Testing: A Survey, pp. 78–105. Springer, Berlin (2009). doi:10.1007/978-3-540-95888-8_4
7. De Francesco, A., Napoli, C.D., Giordano, M., Ottaviano, G., Perego, R., Tonellotto, N.: Midas: a cloud platform for soa testing as a service. Int. J. High Perform. Comput. Netw. **8**(3), 285–300 (2015). doi:10.1504/IJHPCN.2015.071254.
8. De Francesco, A., Di Napoli, C., Giordano, M., Ottaviano, G., Perego, R., Tonellotto, N.: A soa testing platform on the cloud: The midas experience. In: 2014 International Conference on Intelligent Networking and Collaborative Systems (INCoS), pp. 659–664 (2014). doi:10.1109/INCoS.2014.62
9. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Pearson Education India, Karnataka (2005)
10. Github. https://github.com/
11. Google Cloud Platform—App Engine. https://cloud.google.com/appengine/
12. Herbold, S., Francesco, A.D., Grabowski, J., Harms, P., Hillah, L.M., Kordon, F., Maesano, A.P., Maesano, L., Napoli, C.D., Rosa, F.D., Schneider, M.A., Tonellotto, N., Wendland, M.F., Wuillemin, P.H.: The midas cloud platform for testing soa applications. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pp. 1–8 (2015). doi:10.1109/ICST.2015.7102636
13. Herbold, S., Harms, P., Grabowski, J.: Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. J. Softw. Tools Technol. Transf. (STTT) (2016)
14. Hillah, L.M., Maesano, A.P., De Rosa, F., Kordon, F., Wuillemin, P.H., Fontanelli, R., Di Bona, S., Guerri, D., Maesano, L.: Automated and intelligent scheduling of distributed system functional testing. J. Softw. Tools Technol. Transf. (STTT) (2016)
15. International Software Testing Qualifications Board (ISTQB): Standard glossary of terms used in Software Testing, Version 2.1 (2010)
16. Mell, P., Grance, T.: The NIST definition of cloud computing. Special Publication 800–145. National Institute of Standards and Technology, U.S. Department of Commerce (2011). doi:10.6028/NIST.SP.800-145
17. Mesnier, M., Ganger, G.R., Riedel, E.: Object-based storage. IEEE Commun. Mag. **41**(8), 84–90 (2003). doi:10.1109/MCOM.2003.1222722
18. Model and Inference Driven Automated testing of Services architectures (MIDAS). http://www.midas-project.eu/
19. Service Component Architecture (SCA). http://www.oasis-opensca.org/sca/
20. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Pearson Higher Education, Karnataka (2004)
21. Test@Cloud. http://www3.informatik.uni-erlangen.de/EN/Research/Test@Cloud/
22. Testing Technologies. http://www.testingtech.com/
23. Testing Technologies: Ttworkbench. http://www.testingtech.com/products/ttworkbench.php
24. Travis ci. https://travis-ci.org/
25. W.W.W.C. (W3C): State chart xml. https://www.w3.org/TR/scxml/