Int J Softw Tools Technol Transfer (2017) 19:309–324
DOI 10.1007/s10009-016-0437-y

Combining usage-based and model-based testing for service-oriented architectures in the industrial practice

Steffen Herbold[1] · **Patrick Harms**[1] · **Jens Grabowski**[1]

Published online: 21 September 2016

Abstract Usage-based testing focuses quality assurance on highly used parts of the software. The basis for this are usage profiles based on which test cases are generated. There are two fundamental approaches in usage-based testing for deriving usage profiles: either the system under test (SUT) is observed during its operation and from the obtained usage data a usage profile is automatically inferred, or a usage profile is modeled by hand within a model-based testing (MBT) approach. In this article, we propose a third and combined approach, where we automatically infer a usage profile and create a test data repository from usage data. Then, we create representations of the generated tests and test data in the test model from an MBT approach. The test model enables us to generate executable Testing and Test Control Notation version 3 (TTCN-3) and thereby allows us to automate the test execution. Together with industrial partners, we adopted this approach in two pilot studies. Our findings show that usage-based testing can be applied in practice and greatly helps with the automation of tests. Moreover, we found that even if usage-based testing is not of interest, the incorporation of usage data can ease the application of MBT.

Keywords Usage-based testing · Model-based testing · Usage monitoring · Web service testing · TTCN-3

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

Patrick Harms
harms@cs.uni-goettingen.de

Jens Grabowski
graboswki@cs.uni-goettingen.de

[1] Institute of Computer Science, Georg-August-Universität Göttingen, Göttingen, Germany

1 Introduction

One of the major challenges in quality assurance is the efficient spending of available resources to get the largest benefit out of them. One strategy is to put the user-experienced quality at center stage through usage-based testing [26]. The idea is to focus the quality assurance on the highly used parts of a system under test (SUT), and, thereby, assure that the core functionalities often required by the users are well tested. The core of usage-based testing are usage profiles. A usage profile is a stochastic description of the utilization of a SUT. The usage profile obtains information on which *events* are triggered by the users. Within this article, we consider the testing of service oriented architectures (SOAs). Hence, the events which model the usage of the SUT are the requests and responses exchanged between services within a service orchestration.

Within the current state of the art, the usage profiles are either modeled completely manually (e.g., [15,17,44]) or automatically inferred from observed usage data of the SUT (e.g., [23,36,37]). In case of manual modeling, the usage profiles can directly be embedded in an environment for further test automation (test generation, test execution,…) like it is done within the tool MaTeLo [15]. However, such manual modeling requires a lot of effort. Moreover, with manual modeling, the test designer must make assumptions about the SUT's usage to create the usage profile. The assumptions might not reflect reality accurately, which would lead to an inaccurate usage profile. In case the usage profiles are inferred from usage data, it is guaranteed that the usage profile reflects the real usage accurately. Moreover, there is no manual effort involved in the creation of the usage profile. However, it is a hard task to embed an automatically inferred usage profile into other tools and modeling languages to allow further automation of the testing. To the best of our knowledge, no such approach exists.

However, when it comes to the practical application of usage-based testing, the combination of the integration with tooling for further test automation with the advantages of automated inference of usage profiles is the best solution. Within this article, we present an approach for the combination of usage-based testing based on automatically inferred usage profiles with model-based testing (MBT) that we developed as part of the MIDAS project [30]. With our approach, we focus on SOA applications. We start with the monitoring of the SOA application to obtain usage data. To this aim, we defined a new usage monitoring approach for SOAs that is easy to integrate and does not slow down the observed system. From the usage data, we create a test data repository and infer a usage profile. Then, we create interaction sequences from the usage profile. We translate these interaction sequences including the related test data into test cases in the MIDAS domain specific language (DSL), an MBT modeling language based on unified modeling language (UML) and the UML testing profile (UTP). The MIDAS DSL allows for the definition of concepts like test contexts, where the setup of the SUT is described as well as interface and data type descriptions of the services within the SUT. The generated tests are fully compliant with the MIDAS DSL and can, therefore, fully harness testing features offered to DSL models, including automated generation of executable Testing and Test Control Notation version 3 (TTCN-3) code. Hence, we demonstrate with our approach a complete end-to-end cycle for usage-based testing that starts with the monitoring of the SUT, then infers a usage profile, from which we can generate tests, which can automatically be executed against complex SUTs.

We evaluate our approach based on two pilot studies from different domains that offer different challenges on the usage-based testing. One pilot has a complex protocol, and the other complex data types. Through the pilots, we evaluate the capabilities of our testing approach and the readiness for use in industry. From these studies we gained valuable insights into practical considerations for usage-based testing and the combination of usage-based testing and MBT. In summary, the contributions of this article are the following:

- A generic and easy-to-integrate usage monitoring facility for SOAs.
- The creation of a test data repository from usage data.
- The combination of usage-based testing with MBT through the generation of tests in the MIDAS DSL, which can then be transformed into executable TTCN-3 from usage profiles.
- An evaluation of the approach from an industrial point of view based on two pilot studies.

The remainder of this paper is structured as follows. In Sect. 2, we discuss the work related to ours. Then, we introduce our approach for the usage-based testing of SOA orchestrations in Sect. 3. Afterward, we consider the most praxis-relevant aspects of our implementation of the approach in Sect. 4. In Sect. 5, we introduce the results of two pilot studies in which we investigate the feasibility of our approach in the industry. Within the discussion of our pilot studies, we also address the threats to the validity of our findings.

2 Related work

In this section, we discuss the literature related to our work. We split the discussion into two parts: the monitoring of SOA, which is platform specific and a general and platform-independent discussion of the related work on usage-based testing.

2.1 Monitoring of SOAs

Monitoring of SOAs is a topic that emerged together with the development of SOAs themselves for debugging purposes. Dan et al. [48] provide categories for monitoring approaches. They divide between *hard-coded* in the services, *soft-coded* through, e.g., code injection and *agnostic-coded* which is external to the services. Our monitoring approach is external to services and, hence, belongs to the latter category.

Similar to our approach, Dan et al. [48] propose a further approach and category being between soft coded and agnostic coded. For this, they utilize proxies deployed in the same application context as the SOA to be monitored. We also utilize proxies, but they can be in separated application contexts and are more loosely coupled to the services.

Chen et al. [10] identified the properties required to be able to monitor SOA applications with respect to the identification of workflow executions. These properties are *service id*, *interface id*, *process id*, and *sequence id*. The first two are used to identify the communication partners. The second represents information about an executed workflow. As Chen et al. note, the information about the communication partners can easily be retrieved and is often supported by SOA frameworks. Information about the executed workflows instead is harder to identify [31]. In our approach, we monitor both the communication partners and executed workflows. We show that for the communication partners, a host name is not sufficient. In addition, we show a solution for monitoring workflows that does not require changes of the SIT which is usually done in other approaches.

2.2 Usage profile and usage-based testing

Within this section, we only consider the literature with a direct relation to software testing. The literature that describes how usage information is exploited for other purposes is not discussed and we refer interested readers to appropriate literature reviews instead, e.g., [27,34].

The notion of usage profiles for quality assurance was first introduced by Littlewood [29] and Cheung [11] for reliablity estimation. They modeled the probabilities that a component is called. For test generation, this concept was picked up by Whittaker et al. [43,44], Walton et al. [40], and Wesslén and Wohlin [42] for test generation. Woit [45,46] used a similar approach, but proposed a different model that allowed restrictions on which events are possible. All these approaches model the usage profile manually. Additionally, the models are simple graph structures and not based on modern modeling languages like UML.

Tonella and Ricca transferred the approach of usage-based testing to Web applications [36–38]. They provide a fully automated approach that also includes usage profile inference from recorded usage. In comparison to our work, Tonella and Ricca focus only on Web applications, whereas we consider event-driven software in general. Furthermore, they use simple graph-based models, which may contain inaccuracies, whereas we support UML models that accurately model the SUT's behavior.

An approach for the accurate inference of the structure of a usage profile has been proposed by Dulz and Zhen [15], Le Guen et al. [28] and Feliachi and Le Guen [17]. The resulting tool MaTeLo [7] allows extending state machines with probabilities to generate test cases based on the modeled usage. In comparison to our work, MaTeLo does not allow the inference of probabilities from usage data. Instead, the probabilities are, e.g., derived from requirements.

A different way to harness usage profiles for testing than just the generation of tests was proposed by Tonella et al. [39]. They assume that if the test model is only an approximation of the real SUT, e.g., because they are automatically created, some of the generated tests are potentially infeasible and must be removed. They propose that if usage information is used to rate the generated tests and select those that match the usage best, it leads to a higher likelihood of only selecting feasible test cases. They use n-grams for their scoring and show that the usage data indeed helps with selecting feasible test cases from automatically inferred models.

Herbold et al. [24] describe how usage-based testing can be performed in an abstract and platform-independent way. They propose a layered model in which the usage-based testing works on abstract events and a translation layer works as intermediate between the testing and concrete platforms. The approach was later implemented in the tool AutoQUEST [25]. AutoQUEST was also used as foundation for the usage-based testing presented in this paper and extended with the required capabilities to support SOAs and the MIDAS DSL.

We believe that all of the above approaches may be extended for SOA testing if they are properly extended. The greatest challenge will be the inclusion of the data complexity of SOAs, which none of the above addresses. Within this article, we describe how AutoQUEST was extended for SOAs and thereby describe the steps that other approaches must consider when extending usage-based testing to SOAs.

3 Approach

Our approach for usage-based testing consists of six steps, which are visualized as rectangles in Fig. 1. Some steps are split up into multiple rectangles in the figure. The parallelograms in the figure are the artifacts produced by one step and the input of other steps. The steps are:

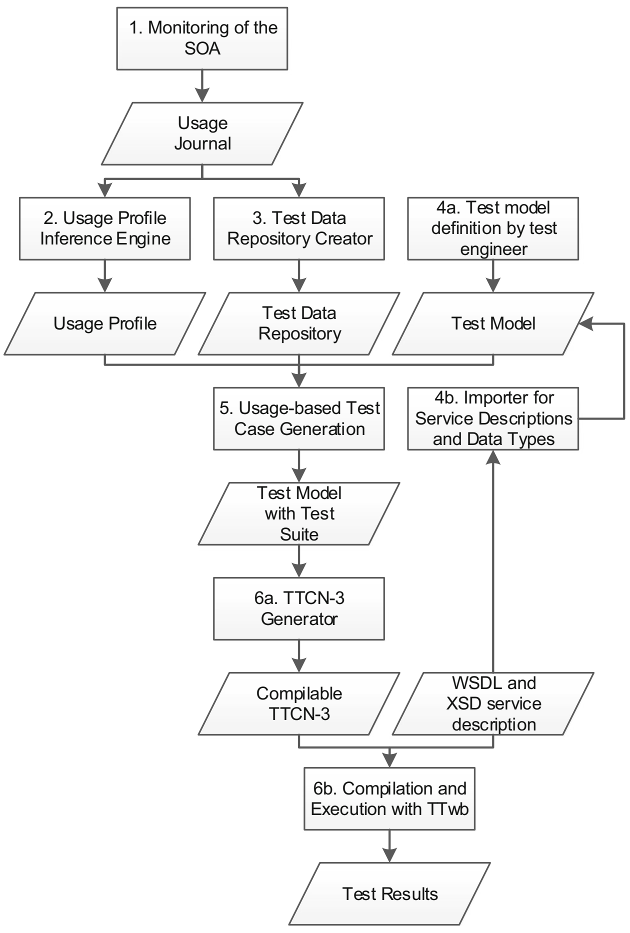
1. Monitoring of the SOA
Usage Journal
2. Usage Profile Inference Engine
3. Test Data Repository Creator
4a. Test model definition by test engineer
Usage Profile
Test Data Repository
Test Model
5. Usage-based Test Case Generation
4b. Importer for Service Descriptions and Data Types
Test Model with Test Suite
6a. TTCN-3 Generator
Compilable TTCN-3
WSDL and XSD service description
6b. Compilation and Execution with TTwb
Test Results

Fig. 1 Approach for usage-based testing of SOAs. The rectangles represent tasks to be performed either by software or manually, and the parallelograms represent artifacts that are created

1. Monitoring of the SOA: the SUT is monitored during its execution to obtain a usage journal.
2. Usage profile inference: a usage profile is automatically inferred from the information in the usage journal.
3. Test data repository creation: we utilize the information in the usage journal to create a repository with test data that can be used for the test generation.
4. Test model definition: a test model of the SUT structure is defined in the MIDAS DSL. Partially, the definitions are imported from existing Web Service Description Language (WSDL) and XML Schema Definition (XSD) files of the SUT.
5. Test generation: generation of tests in the MIDAS DSL with the usage profile as an extension of the test model.
6. Test execution: being a combination of generation, compilation, and execution of TTCN-3 code from the test model for the execution of the generated tests.

In the following subsections, we describe each of these steps and explain how the steps interact with each other.

3.1 Monitoring of SOAs

SOAs consist of several services performing an orchestration. A service offers functions that can be called by another service. We refer to the service offering a function as a *server*. The service calling a function is a *client*. Typically, functions are provided as calls with parameters and return values. For monitoring interactions of services in an SOA, we monitor function calls of clients on servers. This includes storing information about the client, the server, as well as parameter values and return values of the function calls.

For monitoring function calls, we utilize an approach based on proxies located in front of any server in an SOA. Any function call of a client on a server is routed through a proxy. The proxy receives a function call, forwards it to the server, receives the results, sends the result back to the client, and logs the request and the result. The basic approach of utilizing proxies for monitoring is shown in Fig. 2. In this figure, the components of a simplified SOA are the gray boxes.

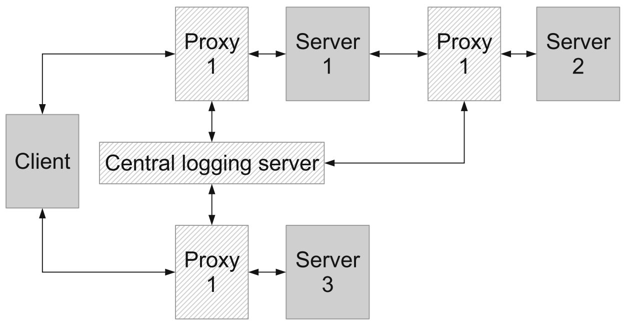

Fig. 2 Example of monitoring a SOA

The hatched components between the SOA components are proxies. If the client calls a function on, e.g., *Server 1*, then this call is routed through *Proxy 1* which stores the corresponding request and response. The advantage of this kind of monitoring is that neither the server's nor the client's implementations need to be adapted. Only the SOA setup, which is usually done by configuration of the services, needs to take the proxies into consideration so that requests are sent to the proxy instead of sending them directly to the server.

In SOAs, typically one client calls several servers and performs the actual orchestration. This also implies that there are several proxies for different servers. However, to create a single log file for complex orchestration, our proxies can be configured to send decentrally recorded function calls to a central logging server. This is also shown in Fig. 2. There, the proxies send the logged exchanges to a central logging server which writes the log file.

The log file stored by the central logging server is in an XML format. It contains entries, one for each recorded function call. We call these entries *exchanges*. Any exchange consists of information about the client, the server, the function call, the parameters, and the return values. This allows to trace all information exchanged between a client and a server for a specific function call.

For some function calls, a server can become a client of a further server. An example is shown in Fig. 2. There, *Server 1* may call *Server 2* to handle a specific call of the client. As our proxies only log an exchange after the result has been received, in such a scenario the exchange between the client and *Server 1* would be logged after the exchange between *Server 1* and *Server 2*. To overcome this misordering, we include in any request and response an *ordering ID*. As this must be unique for all recorded requests and responses, it is retrieved from the central logging server at the time the request or the response is received by a proxy. Through this, we can subsequently identify the order of requests and response in which they were observed by the proxies, but still each function call is a single entry in the log file. Through this, we can also handle ordering issues that may occur due to communication delays between the proxies and the central logging server.

When monitoring function calls, it is important to identify the client that performed a specific function call. In many situations, it is sufficient to know the host from which a request came. But if several clients run on the same host, the host itself is not sufficient anymore. Even the port number of the clients is not helpful, as this usually changes on any function call. To be, however, able to identify clients, we set up several proxies for the same server, each dedicated to a specific client. Afterward, a client calls a server only through its dedicated proxy. Through this, we can identify which client sent a specific function call to a server.

In this paper, we monitor only SOAs being set up using SOAP via hypertext transfer protocol (HTTP). Hence, our proxies are HTTP proxies and record HTTP requests and responses. This may also lead to recording other exchanges than SOAP messages. For example, we also record HTTP requests for retrieving the WSDL of a service.

3.2 Usage profile inference

Once we have the usage data available from the monitoring, we create a usage profile of the behavior of the SUT. For this, the most important part is not the inference of the profile itself, but rather the preprocessing of the usage data. This is due to the fact that the monitored usage data contains multiple sources for noise in the data, which would lead to a bad usage profile if they were not treated.

The first preprocessing step happens directly during the parsing of the usage data. Since the monitor observes an actually deployed version of the SUT, the usage data contains references to the deployed services instead of logical names for the services. For the creation of a good usage profile, we require logical names, otherwise the usage profile would be bound to a specific deployment of the SUT. To this aim, we define a mapping between the paths of the deployed services and the logical names of the services. For example, we map the following path in a service unified resource locator (URL)

```
/ws/WarehouseReferenceImplementation
                Service
```

to the logical service name `warehouseService`. Moreover, we also require to know from which client a call to a service came. To be able to resolve this, we require that each client calls a service on a different port which we achieve by setting up the dedicated proxies. Then, we use the port numbers to identify from which logical client a call came from. For example, if a call to the `warehouseService` was on the proxy port `8087`, we know it came from a service of the type `transportService`. Through this logical mapping, we keep the details of the service deployment out of the usage profiles and internally just have a view on the logical communication that took place, based on the clients that sent requests and the services that received them. In the following, we refer to each logical communication as event.

As further preprocessing, we remove all recorded exchanges that were not SOAP requests, e.g., the WSDL retrievals. Then, we split the exchanges consisting of request and response within the usage data into separate events: one for the request and the other for the response. After splitting, we sort the events using the ordering ID assigned by the monitor. Now we have all service calls in the correct order and the requests and responses are separated from each other. Next, we normalize the method names. Due to the underlying technologies[1], for some service calls a suffix Request/Response is added to the service operations. We remove these suffixes automatically for two reasons. First of all, they would lead to different names for the requests and responses of the service operations, which would make matching of pairs later on more difficult. Second, it would interfere with the generation of tests in the MIDAS DSL later on (see Sect. 3.6). As a final step of the preprocessing, we remove calls to services that shall be ignored in the usage profile and, consequently, in the test generation. This may be due to the fact that the testing shall only be done for a subsystem of the complete SUT. Hence, we drop all calls to or from the ignored services.

Once the pre-processing is completed, the usage profile is derived. The usage profile is stateless and internally contains a Markov model [12,18]. With Markov models, we can model the probabilities of events, given the last events. The number of events that influence the probability of the next event is the Markov order. For example, with a Markov order of one, only the last SOAP message sent influences the next message within the usage profile; with a Markov order of four, the last four messages influence the next message, etc. With increasing Markov order, the internal state of the system is better captured and the likelihood of capturing invalid behavior in the usage profile decreases. On the other hand, a high Markov order decreases the randomness of the usage-based testing, which can be a disadvantage, as many different tests shall be generated. We currently support any order of Markov model and the concrete order can be defined by the test engineer. In addition to the observed events, the usage profile also contains a START and an END event. The START event is prepended to each observed sequence, and the END event is appended to each observed sequence. This way, the usage profile also contains information about the first/last event of a user session.

In Fig. 3, we show an example of a small usage profile we inferred from an actual usage journal of a real world system. The usage profile is a first-order Markov model. The nodes represent the SOAP messages that were sent. The edges denote the probability of the next SOAP message. We deliberately selected a very small and linear example, because the profiles grow rapidly. We only have two operation calls: a call from the `pointOfSaleService` to the `materialSupplierService` of the operation

```
purchaseOrder(orderId,…)
```

and a call from the `materialSupplierService` to the `pointOfSaleService` of the operation

```
purchaseOrderConfirmation(orderId,…).
```

[1] E.g., JAX-WS (https://jax-ws.java.net/) creates for each operation an input action and an output action. The input action gets the suffix Request and the output action gets the suffix Response.

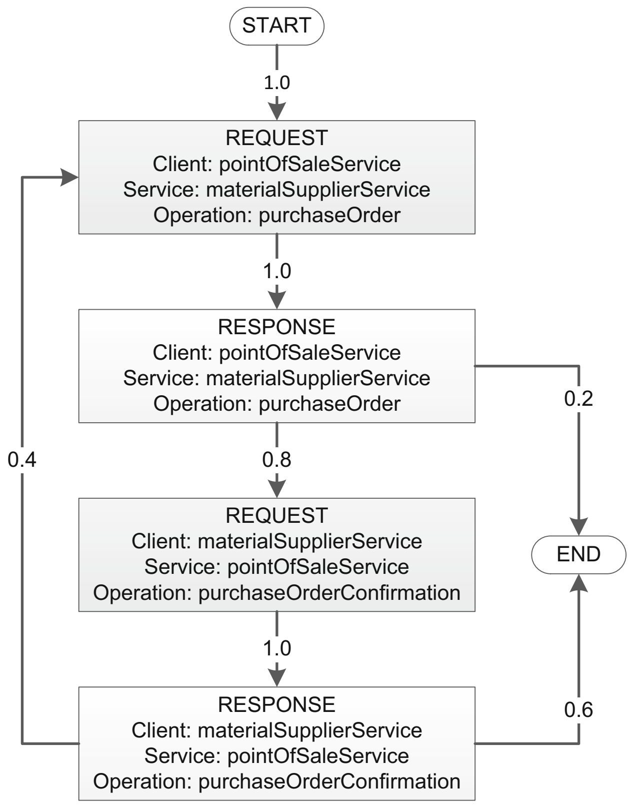
START
1.0
REQUEST
Client: pointOfSaleService
Service: materialSupplierService
Operation: purchaseOrder
1.0
RESPONSE
Client: pointOfSaleService
Service: materialSupplierService
Operation: purchaseOrder
0.2
0.4
0.8
REQUEST
Client: materialSupplierService
Service: pointOfSaleService
Operation: purchaseOrderConfirmation
END
1.0
0.6
RESPONSE
Client: materialSupplierService
Service: pointOfSaleService
Operation: purchaseOrderConfirmation

Fig. 3 Example for a usage profile with a Markov model of order one

For both operations, the request and the response are part of the usage profile as separate events. The probabilities of 1.0 after the requests mean that the response always followed the request[2]. After the responses, we have multiple edges with different probabilities, depending on the likelihood of the next message being sent.

3.3 Test data repository

A problem that frequently occurs with model-driven approaches is that the generation of interactions is simple, but the question remains where the test data comes from. This means that test engineers must either manually define a repository with test data or define a strategy for the randomized creation of valid test data. Through our usage journal, we can offer an alternative to this. Since the usage journal contains the complete interactions between services in the SUT, this automatically means that it also contains the data that were sent with these interactions, i.e., the data that were part of the body of a SOAP request or response.

Our approach for the creation of a test data repository is quite simple: for each combination of client, called service, and service operation, we create a set of all observed SOAP bodies. Then, when we require test data for a SOAP operation, we can simply look up the set of SOAP bodies for the SOAP operation and pick an entity of the test data we desire. The repository itself does not restrict the test data selection strategy and allows, e.g., the selection of always the same entity from the data set (fixed value), a random SOAP body that was observed, and even complex strategies where restrictions on the allowed SOAP bodies are defined, e.g., through XPath expressions.

3.4 Test model definition

A step that can occur in parallel to the usage monitoring, usage profile generation, and test data repository creation is the definition of the test model. The test model contains a formal description of the complete SUT in the MIDAS DSL. The MIDAS DSL is based on UML [33] and UTP [8] and was designed for the specific purpose of SOA orchestration testing as part of the MIDAS European project [30]. It consists of two parts. First, we have the definition of the SUT infrastructure, i.e., the services and the operations they offer as well as the data that is exchanged. Second, we have the definition of test contexts which define the structure of concrete test beds for which tests cases shall be generated. The tests themselves are described as UML interactions.

The first part, i.e., the infrastructure, can be automatically created from the WSDL and XSD descriptions of the SUT [41]. Through model transformation, UML entities for each defined service and data type are created. Without automation of this step, using the MIDAS DSL would be infeasible for larger services and service orchestrations because the manual modeling of the data types in UML would require too much effort. Moreover, manual modeling of complex data types is very error prone in comparison to automated transformation. The second part, i.e., the creation of test contexts, is manual. However, the required amount of work comprises the following steps:

– Definition of logical service components: a UML component for each logical service within the SUT is created and the ports that the service offers for communication are defined. The ports reference the service interfaces that are automatically created during the first part of the DSL model creation.
– Creation of a test context: a UML component diagram that serves as the test context is created. The UTP TestContext stereotype is applied to the main component to formally define the role of the component in the model.
– Addition test bed components: the UML components for the services are added to the test context. To each of the test components, the UTP TestComponent stereotype is applied. To each of the services, a test to the SUT stereo-

[2] Please note that this does not have to be the case in synchronous communication if a service internally sends a request to another service before responding to the client.

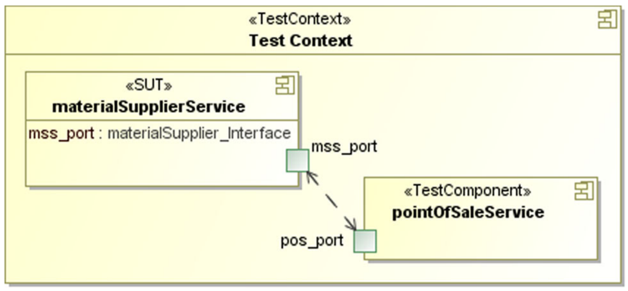

Fig. 4 Example for a test context with two services, one with the role of a test component that drives the testing and the other as the SUT

type is applied. There must be at least one component with the TestComponent stereotype in each test context to drive the testing.

– Addition of connectors: between the ports offered by the components, connectors are added to create the communication infrastructure within the SUT.

Figure 4 shows an example for a test context with two services: `pointOfSaleService` as test component and `materialSupplierService` as SUT. The services both offer a port by which they are connected for communication via the exchange of message.

Once the test context is created, it can be extended with test cases. The test cases consist of two parts: a UML operation to which the UTP TestCase stereotype is applied and a UML interaction that implements the logic of the test case. Moreover, in case the messages that are created for the UML interaction require test data, UML instance specifications for the required data must be defined. It is possible to define such tests manually. However, this is out of scope of this article. Instead, we create the test cases automatically from our usage journal.

3.5 Test generation

The test generation consists of two parts. First, we create a sequence of interactions between services within the SUT. Then, we create a representation of these interactions as test case within a test context in the MIDAS DSL.

The foundation of the creation of a sequence of interactions between services is the usage profile. We create the sequences by randomly walking the usage profile. This means, that we start with our START event and then randomly draw the next event based on the probability we observe in the usage profile. Since the events are the interactions between services, we now have the first service call of our test case. Then, we continue to draw events based on the probability in which they occur. We get this probability also from the usage profile and thereby create our complete interaction. We finish when we either reach the END event or a user-defined number for the maximal test case lengths.

Due to the random nature of the test generation, it may be possible that invalid test cases are generated. There are two ways a test case can be invalid. The first is that we stopped with the generation at an inappropriate point of time. For example, we did not yet generate the response event for an already created request event. We can automatically detect such inconsistencies by simply matching request/response pairs. We resolve any problems we find by simply dropping the incomplete communications from the sequence of interactions. The second way is that the service logic of the SUT orchestration is broken. For example, a service A might first expect a call to operation X and then a subsequent call to operation Y. Because the complete internal state of the SUT is not known in the usage profile and it is only estimated through the Markov order, we might generate a sequence where we call operation Y without calling X first. Such invalid test cases can only be excluded through manual analysis of the sequences and/or the execution of tests. However, by choosing a high Markov order, a test engineer can avoid such inconsistencies. We elaborate on the choice of a concrete Markov order in our case studies (see Sect. 5).

Once the test sequences are created, they are converted into MIDAS DSL compliant UML interactions in a test context, with one interaction for each test sequence. The generation of the test case in the DSL consists of two parts. We generate an operation with the UTP TestCase stereotype and a UML interaction that defines the messages that are sent as part of the test case. To this aim, we first have to create lifelines for each component in the test context. The lifelines represent the services in the SUT. Then, we create a message for each event in our test sequence. We determine the source and target lifelines of the messages from the client name and called service name, respectively. Similarly, we determine the service operation in the UML model via its name in the interfaces that are provided by the lifeline of the called service. Figure 5 shows an example of a very simple test case where two messages are exchanged, including their response. First, the `pointOfSaleService` calls the `purchaseOrder` operation of the `materialSupplier` and receives the response to this call. Then, the `materialSupplier` calls the operation `purchaseOrderConfirmation` of the `pointOfSaleService` and receives the response.

From the operation, we can also see the signature of the method, i.e., the required test data. In case we are creating the message for a request, we only set values for all parameters that are marked as input or input/output parameters. All other parameters are set to the UML literal NULL to signify that they are empty. Similarly, in case we are creating the message

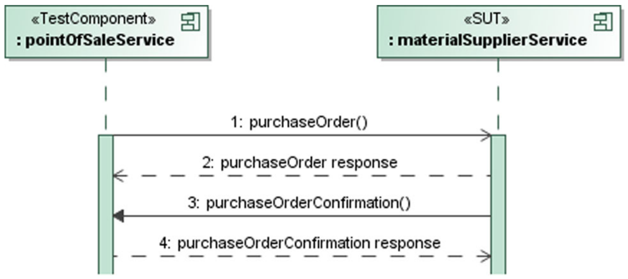
«TestComponent»
: pointOfSaleService
«SUT»
: materialSupplierService
1: purchaseOrder()
2: purchaseOrder response
3: purchaseOrderConfirmation()
4: purchaseOrderConfirmation response

Fig. 5 A UML sequence diagram that shows the UML interaction of a simple test case where two messages are exchanged, including their responses

Listing 1 Example for the body of a SOAP.

```
<S:Body>
  <purchaseOrder>
    <purchaseOrderXML>
      <idOrder>order−18gog</idOrder>
      <idProduct>POTATOES</idProduct>
      <quantity>5000</quantity>
    </purchaseOrderXML>
  <purchaseOrder>
</S:Body>
```

for a response, we only set values for all parameters that are marked as output or input/output parameters and set the other to the UML literal NULL.

The assignment of the values to the parameters of the operations is the second major part of the test generation. To be able to assign values to messages within UML interactions, we require UML instance specifications for the data type of the operations parameter. An instance specification can be considered as the UML equivalent of an initialization of a data type. UML instance specifications can be added as the values for messages in UML interactions. We explain our concept through an example. Figure 6 shows a data type including its nested sub-type on the left as well as the respective instance specifications on the right, which we will now use for the explanation of our approach.

Within our models, we have different kinds of data types: primitive types, enumerations, and complex data types. The primitive types are, e.g., strings and integer and their values can be set directly. In our example, the type `orderMessageType` has internally two strings and one integer, which are primitive types. The instance specification `purchaseOrderXML_Instance` directly contains the values of the strings and the integer. The same holds true for enumerations, which can have a fixed list of values that can be directly set. Complex data types are more difficult. Here, we require a separate instance specification for each nested sub-type. In the example, we see that within the instance specification `purchaseOrder_Instance` for the type `purchaseOrder`, the value of the attribute `purchaseOrderXML` is the instance specification `purchaseOrderXML_Instance`.

To generate the values within the instance specifications, we harness our test data repository. It contains the bodies of the SOAP requests that fit the messages. Hence, we can find the values for the parameters within the body. Listing 1 shows the body of a SOAP request associated with the above data. We create the instance values by mapping the elements of the SOAP body to the elements of the UML model. Every time we encounter an attribute that is a complex type, we create the appropriate instance specification for the complex type and recursively continue with the creation of instance values. When we encounter an enumeration or primitive type, we set the values directly, as we infer them from the matching element in the SOAP body.

3.6 Test execution

After the extension of the test model with the interactions for the test cases including all test data, we go from the test generation part to the test execution part of our approach. As back-end for the test execution, we use TTCN-3 [19]. Hence, the first step is the transformation of the test model in the MIDAS DSL into TTCN-3. Here, we utilize an approach based on a model-driven architecture (MDA) framework. This framework is already used to import the service descriptions from the WSDLs and XSDs into the MIDAS DSL. The test descriptions are on the Platform Independent Model (PIM) layer of the model. Using model transformations, the

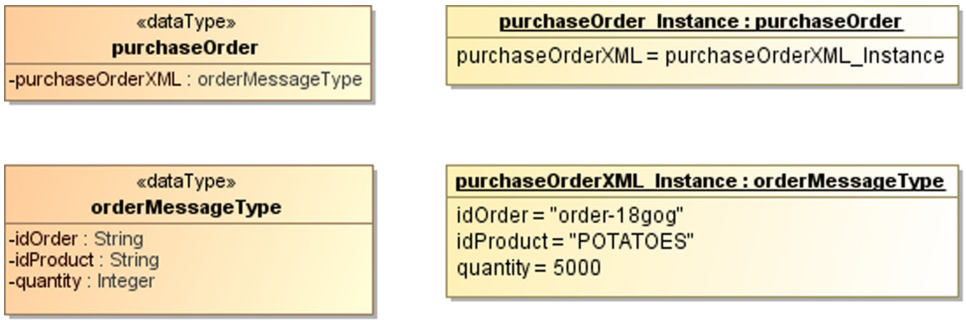

Fig. 6 Example for UML data types and their respective UML instance specification in the MIDAS DSL

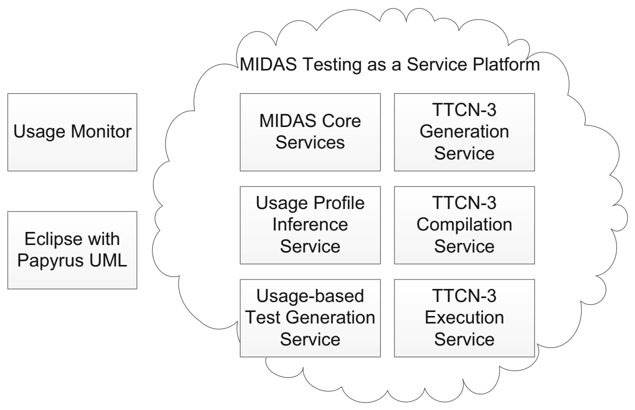
MIDAS Testing as a Service Platform
Usage Monitor
Eclipse with Papyrus UML
MIDAS Core Services
TTCN-3 Generation Service
Usage Profile Inference Service
TTCN-3 Compilation Service
Usage-based Test Generation Service
TTCN-3 Execution Service

Fig. 7 Overview of the components provided to support the usage-based testing with MIDAS

MDA framework automatically translates the test model into TTCN-3 that is compatible with the codecs and adapters for WSDLs and XSDs provided by the tool TTworkbench [35]. The complete model transformation approach by the MDA framework that is implemented for the MIDAS DSL is out of scope of this article and explained in detail in [41]. We use the TTthree and TTman components of of TTworkbench for the compilation and execution of the generated tests and collect the test result for presentation to the users.

4 Implementation

Since we are concerned with the applicability of our approach in practice, we now give a brief overview over the tooling we provided by the MIDAS project. Figure 7 shows the most important components of our approach. MIDAS provides a cloud platform that we refer to as testing as a service (TaaS). Our tooling is divided into two parts: components running outside the MIDAS TaaS platform and components on the platform. Outside of the platform, we have the usage monitor which is described in Sect. 3.1 and Eclipse Papyrus UML editor [16] for the creation of MIDAS DSL models. Papyrus UML is based upon the Eclipse UML framework. Within MIDAS, we work with the Kepler release of Eclipse UML. We cannot support other UML editors in our implementation due to inconsistencies between XMI documents used to store and exchange UML models created by various UML tools.

As part for the MIDAS platform, we have several services available to the users. The MIDAS core services provide functionalities like file management, user management, and the execution of other services like the usage profile inference service. Details on the core services and the underlying cloud platform are found in [13,14]. As we describe in Sect. 3.6, we use TTworkbench for the compilation and execution of the tests. The TTCN-3 compilation service and the TTCN-3 execution service are wrappers for the execution of the respective TTworkbench functions on the cloud platform. The licensing of TTworkbench is handled by the MIDAS platform. Details on this are discussed in [14].

The components related to usage-based testing, i.e., the usage monitor, the usage profile inference service and the usage-based test generation service are all based on AutoQUEST [25], a tool suite for usage-based analysis of software independent of the platform. AutoQUEST was extended with a monitor for HTTP communication which can be utilized for usage monitoring of SOA applications. The created usage journals can be processed by AutoQUEST and are compliant with the abstract representation of events required by AutoQUEST. Based on this representation, we extended AutoQUEST with the required processing steps discussed in Sect. 3.2 to provide a clean usage journal as foundation for the usage profile inference. AutoQUEST natively offers various usage profiles, e.g., Markov models of any order as well as the generation of event sequences from usage profiles according to the defined probabilities. A further extension to AutoQUEST we provided is the extension with a test data repository for the monitored service usage data, as described in Sect. 3.3. Moreover, we added the capability to create MIDAS DSL compliant test cases from event sequences consisting only of the client name, called service name, operation name, and whether it is a REQUEST or not (see Sect. 3.5 for an explanation of the approach).

5 Pilot studies

To prove the feasibility of our approach to industrial applications, we applied it in pilot studies in two real-world SOA applications. The studies have the aim to provide insights into usage-based testing from an industrial point of view. Within this section, we first define the concrete objectives we investigate with the pilot studies. Then, we describe the methodology followed in the pilots. Afterward, we discuss for both pilot studies the concrete SUT and the results achieved with the usage-based testing.

5.1 Objectives

With our pilot studies, we hope to get feedback from the industry on the following issues:

– the feasibility as well as the usefulness of usage-based testing in practice,
– the usage data collection in a privacy critical domain, and
– the applicability in domains with complex data types and complex protocols.

5.2 Methodology

For the pilots, we worked together with industrial partners. In the following, we describe how we implemented the usage-based testing in our pilots. For each step, we detail if the industrial partners executed the step by themselves or how we supported them.

The first step is the installation of the usage monitoring in the reference implementations of the SUTs. The pilots could perform this without trouble with the monitoring tool we gave them. There were some discussions between us and the industrial partners on the best setup for monitoring proxies. While we provided the input, the final decisions were made by the pilot partners. Moreover, the pilot partners were also responsible for the installation and setup of the monitoring within their environment, without intervention from our side.

For the creation of the test models, we provided the partners with the imported libraries and a user guide on how to create test models. Moreover, we conducted two face-to-face training sessions to teach the usage of the MIDAS DSL. Afterward, the industrial partners created the test models, i.e., the test context, ports, etc. on their own.

For the remaining steps of the approach, i.e., usage profile inference, test generation, TTCN-3 generation, and TTCN-3 execution, we gave the industrial partners a simple running example, together with an explanation of how the example works. The execution of tests and the adoption of the examples to their needs were performed by the industrial partners on their own. However, we were always available to answer further questions and give suggestions on how to apply our approach.

5.3 Systems under test

In this article, we consider two pilot systems for our studies, which we discuss in the following.

5.3.1 Supply chain management pilot

The pilot system is a supply chain management (SCM) system based on the GS1 LIM standard [20] for the interoperable implementation of services within a supply chain. Our partner for performing a pilot study was ITAINNOVA [3], a research institute from Spain with a close relationship to the industry. The pilot does not implement the complete GS1 LIM standard due to its overall complexity. Instead, only services for four types of participants in a supply chain are used:

- a material supplier service,
- a transport service,
- a warehouse service, and
- a point of sale service.

The services interact with each other to perform operations within a supply chain. For example, the point of sale can order materials from a material supplier and the shipment of the materials is organized by contacting a transport service. Overall, with just the four service types, the number of possible exchange sequences is already exploding. In our pilot setup, we only consider one service of each type, i.e., one material supplier, one point of sale, and so on. With more instances of each service, the complexity would soon become overwhelming, which would be counterproductive for an initial evaluation of the capabilities of the usage-based testing. To give an impression of the complexity of the underlying protocol within the pilot, Figure 8 shows just the protocol of the transport service on its own, without even taking the other services into account.

The data types of this pilot are of medium complexity, because the pilot uses a slimmed down version of the GS1 LIM standard. Due to these modifications, no new primitive types are introduced, e.g., by defining restrictions on the range of an integer or the values of a string. Moreover, only multiplicities of exactly one are used for the data types and there are no optional elements or choices.

5.3.2 Health care pilot

The second pilot is a health-care system based on standards from the Health Level Seven International (HL7) [2] Healthcare Services Specification Program (HSSP) [22] for the creation of interoperable solutions for management of patients and their data related to health care. The pilot exposes only two of the HSSP standard services to be tested with MIDAS, even if four of them have been implemented in the overall platform. The standardization follows an MDA approach, i.e., Computation Independent Model (CIM), PIM, and Platform-Specific Model (PSM) layers are defined for all services. Both services used in the pilot are a joint specification effort by HL7 for the CIM layer and the OMG [5] for the PIM and PSM layers. Our second partner for performing this pilot study was Dedalus S.p.A. [1], a provider of health-care systems.

The first implemented service specification is the Retrieve Locate Update Service (RLUS) [32]. Its scope is the harmonization of how information resources are stored, retrieved and updated, independently of their nature and format in a cross-organizational framework. RLUS exposes two service interfaces within our pilot. The *RLUSManagementAndQueryInterface* to manage the persistence and retrieval of health-care data and the *RLUSMetadataInterface* for the definition of the so-called semantic signifiers, a construct of the standard that defines the meaning of the exposed data.

The second standard is identity cross-reference service functional model (IXS) [21], an HL7 standard for

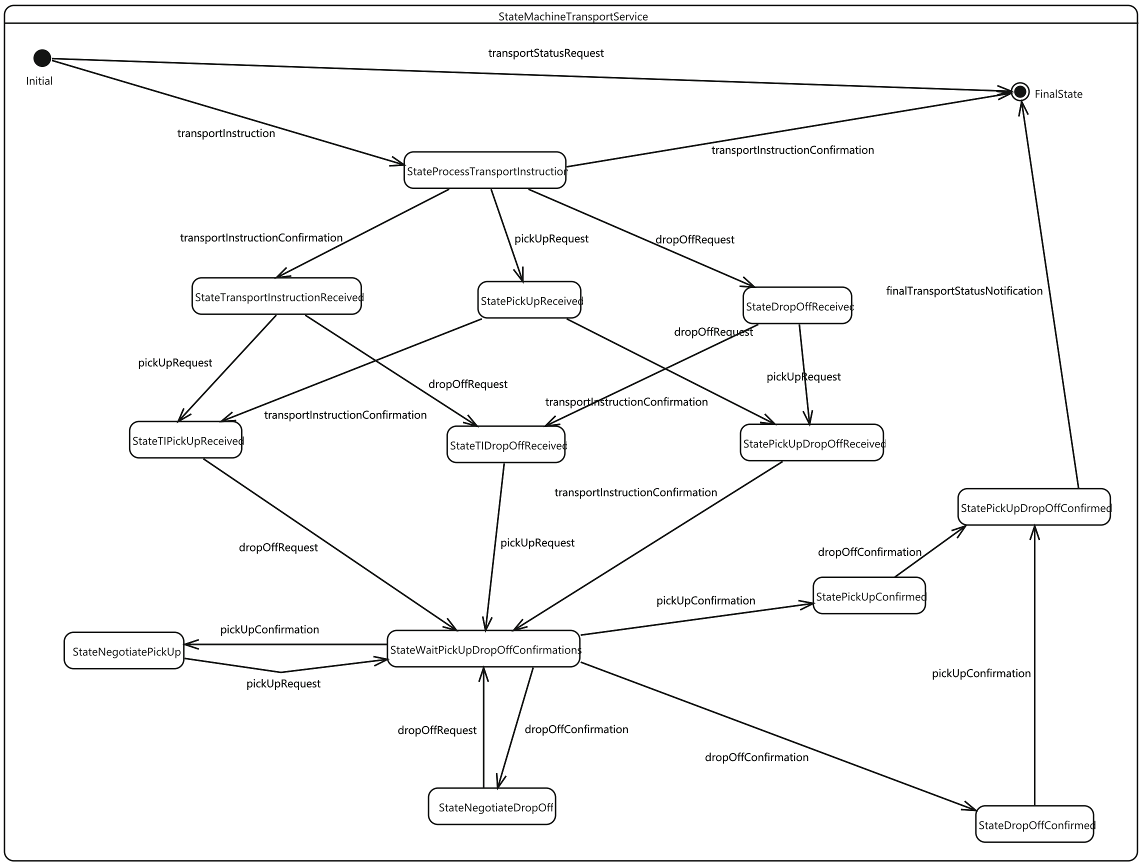

Fig. 8 State machine that describes the behavior of the transport service in the supply chain management pilot

the identification of entities within the health-care system (e.g., patients, health-care provides, devices and so on). The standard defines how demographic and other identifying characteristics called *traits* are used to resolve unique identifiers. Within our pilot, IXS exposes three service interfaces: the *IXSManagementAndQueryInterface* for the resolution of identifiers; *IXSAdminEditorInterface* for the administration of identifiers (e.g., merging of identities); and the *IXSMetadataInterface* for the definition of semantic signifiers that define the meaning of the concrete identifiers.

From a testing point of view, this pilot is the opposite of the SCM pilot. Both RLUS and IXS are basically standards for create/read/update/delete (CRUD) operations, which means that there is no complex protocol involved. However, the data types involved are extremely complex and utilize nearly the full flexibility offered by XML Schema, including choices, nested constructs of sequences and choices, restrictions on primitive types, and even the ANY construct that allows an arbitrary element. This is made even more complex by the fact that the service specifications are generic and different semantic signifiers can be added over time to be able to handle new resources in RLUS and traits in IXS.

5.4 Results

Within this section, we present the result of the application of our approach to our pilot systems, with a focus on the overall feasibility, open problems, and the effort required to implement our approach. We go over the results step by step, following the approach we defined in Sect. 3.

5.4.1 Usage monitoring of the SOA

The collection of usage data was different for both pilots. For the SCM pilot, only a reference test implementation that did not have real users was available. Here, sample scenarios were executed against the reference implementation to provide usage data for us. For the health-care pilot, on the other hand, a real system that received new calls daily was running.

For the SCM, we ran into the issue that several clients were installed on the same host. Hence, the pilots had to install multiple proxies for the same server, each receiving the requests of a different client. But the pilot partner found a setup for this that was easy to adapt if a new version of the monitor is to be installed. For the health-care pilot, we did not observe this issue. This showed that depending on the complexity of the SOA, the initial configuration effort for the monitoring may increase from easy to medium [9], but still the maintenance effort for the proxies is low.

The SCM pilot used a central logging server. The health-care pilot only used a single proxy as all servers were installed in the same application server and there was no communication recorded between the services. In contrast to our expectations, both pilots did not report any problems with the efficiency of the SOAP message processing or a slow-down of the SUT.

The amount of data collected for the SCM pilot was less than 1 MB, since only sample scenarios were executed. For the health-care pilot, we recorded several GB of data over the course of 1 year. The data contained millions of calls to the system. Due to the proprietary nature of the system, we cannot report on the exact number here, but only the scale. The monitoring process did not slow down the system noticeably, even the growing data. Larger scales of data due to more heavily used systems were not considered in our work and may require the usage of big data techniques for scalable monitoring and also the subsequent parsing of the recorded data. This would require a replacement of the XML-based log format, e.g., with an NoSQL database to which the records are written.

5.4.2 Generated usage profiles

The usage profile inference worked fine and without problems. AutoQUEST provided a good and stable foundation for this purpose, with which we could create usage profile with different Markov orders for both pilots without any problems. From our pilot's point of view, the complexity of the usage profile inference was completely hidden away. They only needed to specify two parameters: the Markov order and file containing the usage journal. No detailed knowledge about usage profiles was required, except some hints on how to select the Markov order. Here, our pilots followed our initial proposals and used a Markov order of four (i.e., the last two exchanges consisting of request and response were considered).

5.4.3 Test data repositories

From an implementation point of view, the test data repository worked fine and did not cause any problems. The pilots did not even realize that we were creating a test data repository internally, until we told them. They worked with the assumption that getting the test data for the automatically generated tests was not problematic.

5.4.4 Test model definition

The definition of the test model was the biggest challenge for our pilots. Both pilot providers were not experts on UML and not familiar with the Papyrus UML editor. We performed two 1-day face-to-face training sessions during which we described the complete workflow and identified problems the pilots had with the MIDAS DSL. During these sessions, the key problem identified was that the number of data types was overwhelming and must be imported directly. Once we added support for this and provided the pilots with a UML library that already contained all data types and service interfaces, the creation of test contexts became straightforward and the pilots were able to create the test models on their own.

5.4.5 Test generation

The test generation worked without major problems. The pilots were able to change the number of generated tests cases without trouble by adapting a configuration file. Moreover, they were able to calibrate the testing through defining minimal and maximal lengths for tests. For the generation of the SOAP call sequences, AutoQUEST already provided a stable foundation for us. The generation of UML interactions to represent the tests in the MIDAS DSL also worked, but could cause problems if there were even only very small flaws in the test model or the logical names of the services assigned during the preprocessing of the usage journal. This is due to the fact that we match the services by name to the entities in the test model. Hence, if there are discrepancies in the naming, it is not possible to match the services.

We were able to help our pilots to avoid this problem through the provisioning of a usage-based model validation service that compares the preprocessed information in the usage journal with the test model before the generation of the tests starts. The validation checks the consistency in the naming and gives hints about the possible sources of mismatches which facilitate the quick finding of problems and correction of the test model and/or the configuration of the preprocessing.

Regarding the logic of the test cases, we did not observe many logical inconsistencies within our testing. Due to the controlled data we got for the SCM pilot, no problems were observed. For the health-care pilot, there was no underlying protocol, i.e., the service is stateless anyways. This means that basically all communication is valid, as long as the test data are valid. Since the test data come directly from observed usage, all data we have were valid.

However, we observed limitations of our approach when it comes to the automated generation of test data for the test cases. In our test data repository, we only collect the SOAP bodies of the messages being sent. For the test generation, we need to map the information in the bodies to entities in the test model (see Sect. 3.5). While we were able to support this for nearly all of the data types in both pilots, we found limitations where this is not possible anymore. This led to two limitations with the adoption of our approach in the health-care pilot: the use of ANY and nested structures of sequences and choices.

The ANY construct undermines our general strategy for the generation of instance specifications in the test model from the test data repository: we infer the concrete data type by looking at which data type is expected. If the model allows ANY data type, we cannot automatically decide which data types to choose. This problem is still unresolved and requires further research. However, to our opinion, this may be a general limitation of automated test data generation and can only be overcome if the actual data types could still be mapped unambiguously despite ANY.

The nested sequence and choice structures are a problem for the MIDAS DSL in general. This is due to the fact that such restrictions are not directly supported by UML data types. Hence, an artificial construct using anonymous inner types that represent the sequences and choices was created. Similar to the ANY problem, this also breaks our strategy for the type inference of the test data we generate. Strictly speaking, there is no type for the sequence in the test data. However, we somehow have to handle this in the model, where we suddenly find an anonymous inner type. To be able to still work with the health-care pilot, we currently use a semi-hard coded solution. We look for the keywords sequence and choice within the data types, in case we suspect that a type might actually be an anonymous inner type. If this is the case, we continue to look for the actual type within the children of the type, until we find a match. This solution is not ideal, but working for now.

5.4.6 Test execution

During the test execution, we did not observe any issues that occurred due to the usage-based testing. This does not mean that we did not face significant challenges. Due to the complexity of the data types, the automated generation of TTCN-3 code for the health-care pilot was not fully supported by the prototypical implementation of the MDA approach for TTCN-3 generation [41]. However, this is not a problem of the usage-based testing, but of the generation of automatically executable tests in the underlying MBT approach facilitated by the MIDAS DSL.

With the SCM pilot, the TTCN-3 generation from the tests was not problematic. However, the provided reference implementation was not set up in a testable way at first. The end points for sending exchanges were fixed within the scenario. Hence, the pilot did not communicate with our TTCN-3 testbed as expected, but rather to the hard-coded targets of the calls. For example, if we called the `purchaseOrder` operation of the `materialSupplier` service, according to the protocol, the next thing that should happen is that the `material- Supplier` reacts with a `purchaseOrderConfirmation` call to our testbed, where we simulate a `pointOfSale` service.

However, this `purchaseOrderConfirmation` was not called on our testbed, but on the `pointOfSale` service in the reference implementation. This problem was addressed by defining the end point using the WS-addressing standard [47]. Once these problems where resolved, we could successfully execute ten generated tests against the reference implementation. The rather small number is due to a small usage profile, as there was only sample usage available for the SCM pilot. However, we uncovered 18 defects through our efforts: 6 race conditions, 3 missing threads, 4 hard-coded responses, 2 problems with the port configuration, and 3 unlimited loops [9].

5.5 Discussion

Our results regarding the practical application of usage-based testing to SOA applications are mostly positive. We could define a stable component for the usage monitoring that did not cause any problems over an extended period of time. Moreover, our pilots stated that the installation of the usage monitor was simple and updating it was unproblematic. This means that the first precursor for our approach, the usage data, does not pose problems. The only potential issue raised by one of our pilot partners was that the quality assurance in case of updates to the monitor must be very good, because problems with the monitor can potentially block all communication between services and, thereby, break the complete SOA application. From our point of view as the engineers of the usage monitor, the most problematic part is the identification of callers by using different ports. We hope to be able to change this in the future by instead using WS-addressing. However, the change of our SCM pilot to WS-addressing was very late within our experiments and did not allow the update of the usage monitor with that capability for this study.

The results of combining usage-based testing with a predefined test model, i.e., the MIDAS DSL, was also for the most part positive. While there was trouble during the pilot studies due to bad or changing naming conventions, which caused problems with the matching of services operations to the actual SUTs, these problems were all easily resolved by introducing the usage-based model validation service. Our experience with the test data repository is also very positive. Other partners working with the MIDAS DSL even requested that we create test data for them, because the manual creation of data in the DSL is a very labor-intensive task. We think that this might be the biggest insight we gained from our experiments: the problems with test data can be resolved through such automated imports from usage data.

Similarly, other users of the MIDAS DSL were interested in manually creating tests with the MIDAS DSL. They also found it easier to generate tests with the usage-based test generator and then adapt them according to their wishes and purposes. Through the automated generation of the tests, they also had directly test data available, which they could manipulate if desired.

Regarding the test results, the main challenge with complex SOA orchestrations seems to be implementing them in a way that meaningful tests are possible at all. In our health-care pilot, we did not observe such problems, but we attribute this to the fact that there is no internal communication within the orchestration, just shared databases. Hence, we are technically rather performing tests of single services together, than the testing of an orchestration. With the SCM pilot, we first experienced the problem of the answering locations. However, even after we resolved that, we first needed to adapt our test engine to deal with that appropriately. Once this was done, we still could only validate the functionality at the boundary of the service orchestration and were not able to validate the internal communication within the orchestration. For this, we would have to add additional interceptors between the services in the orchestration that report the internal communications to the test engine. Such interceptors are in principle similar to our usage monitor, but with a different task, i.e., report to the test engine. We could resolve this by creating a new version of the usage monitor that can serve a TTCN-3 test component.

5.6 Threats to validity

We identified several threats to the validity of our study, both internal and external nature.

5.6.1 Internal threats

We were working together with our pilot partners for three years, which means that the pilot systems and parts of our approach were developed in parallel. This may have affected the development of the pilots in such a way that they were not representative of industrial software anymore. However, since both pilots are based on standards and are working with standardized protocols and data types, it is unlikely that such an effect occurred in a way that it changes our findings.

Similarly, the MIDAS DSL was also developed in parallel to our testing approach. The combination of usage-based testing with other MBT approaches might be more difficult, because our requirements for usage-based testing might also have affected the development of the MIDAS DSL. From our point of view, this was not the case, as part of the DSL is based on the already standardized UTP 1.2 and the remainder is used as input for the currently ongoing standardization of UTP 2.0. These standardization influences outweigh any benefits through tailoring the DSL to the needs of usage-based testing. This is exemplified in a way that no real logical identifiers for service types were added to the DSL and we had to match them by name instead.

The obtained usage data were partially due to sample usages of the service orchestrations and not from the field. Our results might change if data from the field were used. This could be especially problematic for the robustness of the test generation which may lead to invalid tests being generated, due to either bad test data or bad capturing of the orchestration protocol.

5.6.2 External threats

We considered only two domains in which SOAs are applied, with the logistics and health-care domain. The considerations regarding usage-based testing might be different in other domains. Moreover, experience with additional pilot studies with partners that do not receive close support from us may lead to different results.

Furthermore, we did not consider larger pilots with a higher amount of exchanged requests. If the approach is applied to larger-scale systems, there may be issues with high amounts of monitored data and, hence, corresponding requirements toward the data analysis and the performance of the monitoring infrastructure.

6 Lessons learned

From our experiences we gathered during the implementation of our approach and due to the adoption of the approach in two pilot projects, we gained three valuable insights into considerations for the application of usage-based testing in practice.

One problem with usage-based testing is that it is hard to make tests executable. The usage profiles only contain logic about the system behavior. Knowledge about how to execute this system behavior is missing. Up till now, we followed the approach to provide a translation from the abstract events directly into a format to be executed by a test driver, e.g., for Java with Jacareto [4], for HTML with Selenium [6], or for Microsoft Foundation Classes (MFC) based applications with a home-brewed test driver [23]. In this article, we present a different approach. Instead of directly generating executable tests, in this case of TTCN-3, we use the MIDAS DSL for MBT as an intermediate representation, from which we can then generate executable tests.

This approach offers two powerful advantages: 1) it enables MIDAS DSL users to harness the test data and test cases generated from the usage-based testing; 2) in case the MIDAS DSL supports the generation of executable tests for a different technology than SOAs, this functionality becomes directly available for usage profiles of this platform. Therefore, the first lesson we learned is the following.

> Combining usage-based testing with MBT is a challenging problem, but yields advantages for both sides.

The second lesson can be seen as a corollary of our first lesson. The biggest problems the pilots had with the MIDAS DSL were related to the test data, due to its complexity. They had problems with the modeling of the data types, which could be resolved through an automated import of the data types from the XSD and WSDL descriptions of the services. Moreover, they had problems with creating instances of the data types in the MIDAS DSL for the same reasons. This problem could be resolved by creating instances of the test data directly from the usage data. Therefore, we learned the following for domains where test data are quite complex and otherwise hard to obtain.

> Usage data are a valuable source for test data.

The third lesson is specifically for the usage-based testing of SOAs. Service orchestrations are already complex, when one just considers their implementation. Due to this complexity, testability is often neglected. This problem seems to be worse for usage-based testing approach. Since we try to automate the testing process as much as possible, we have some requirements regarding the compliance of the SUT, e.g., that we are able to identify from which logical entity (e.g., another service or an end user) a call of a service operation came. Such consideration must be structurally supported by the SUT and are hard to add retrospectively. Similarly, the SUT must not be deployed in a way that our monitor is not installable, e.g., because re-routing the service calls through the monitoring proxy is not possible due to hard-coded information. Hence, we learned the following lesson for the future.

> Concerns regarding usage-based testing should already be considered early during the system development to facilitate a testable and monitorable structure of the SOA. Otherwise, the application of usage-based testing can be infeasible, e.g., due to hard-coded relationships and or the inability to infer the source of a service operation call.

7 Conclusion

Within this article, we discussed the application of usage-based testing in practice. To this aim, we proposed an approach for the application of usage-based testing to SOA. The first part of our approach is a usage monitor for SOA that is non-intrusive and easy to integrate. Then, we use the obtained usage data from the monitor to train a usage profile from which we can generate test sequences. Moreover, we create a repository with test data from the usage data. We combine the usage-based testing with an MBT approach based on the MIDAS DSL, a UML and UTP-based language for the definition of tests. Through this combination, we were able to automatically generate executable tests. We evaluated the approach in two pilot studies, in which we gained valuable insights into the practical feasibility of usage-based testing.

During our pilot studies, we observed some problems that should be addressed in the future. Most importantly, the monitoring of the clients of service calls is currently quite complicated and requires the usage of different ports to identify different callers. Here, a more powerful and flexible way with less effort for the configuration of the SUT would greatly help with the applicability of the method. Moreover, we would like to go further ahead with the integration of usage-based testing with MBT. To this aim, we would like to extend state machines that describe the behavior of the services within the MBT approach with usage information. This way, we can change the models within the MBT environment into usage profiles, which should enable new and improved methods for test generation.

Acknowledgements This work was done in the context of the “Model and Inference Driven-Automated testing of Services architectures” (MIDAS) European project (project number 318786). We would like to thank Testing Technologies for their support in terms of licensing as well as feedback to support requests regarding TTworkbench; Fraunhofer FOKUS for the creation and maintenance of the MIDAS DSL and TTCN-3 generation; and our pilot partners from ITAINNOVA and Dedalus S.p.A. for their support in conducting the pilot studies.

References

1. Dedalus. http://www.dedalus.eu/
2. Health Level Seven International. http://www.hl7.org.uk/
3. Itainnova-instituto tecnológico de aragón. http://www.itainnova.es/
4. Jacareto. http://sourceforge.net/projects/jacareto/
5. Object management group (omg). http://www.omg.org
6. Selenium webdriver. http://www.seleniumhq.org/
7. ALL4TEC: MaTeLo. http://www.all4tec.net/index.php/en/model-based-testing/20-markov-test-logic-matelo(linkcheckedJune2nd, 2014)
8. Baker, P., Dai, Z.R., Grabowski, J., Haugen, O., Schieferdecker, I., Williams, C.: Model-driven testing: using the uml testing profile. Springer-Verlag New York Inc, Secaucus (2007)
9. Barcelona Liédana, M.A., López-Nicolás, G., García-Borgoñón, L.: Practical experiences in the usage of midas in the logistics domain. Software Tools for Technology Transfer (STTT), accepted (2016)
10. Chen, C., Zaidman, A., Gross, H.G.: A framework-based runtime monitoring approach for service-oriented software systems. In:

Proceedings of the International Workshop on Quality Assurance for Service-Based Applications, QASBA '11, pp. 17–20. ACM, New York, NY, USA (2011). doi:10.1145/2031746.2031752

11. Cheung, R.C.: A user-oriented software reliability model. IEEE Trans. Softw. Eng. **6**(2), 118–125 (1980). doi:10.1109/TSE.1980.234477
12. Cover, T.M., Thomas, J.A.: Elements of information theory, 2nd edn. Wiley, Hoboken (2006)
13. 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: Intelligent Networking and Collaborative Systems (INCoS), 2014 International Conference on, pp. 659–664 (2014). doi:10.1109/INCoS.2014.62
14. Di Napoli, C., De Francesco, A., Giordano, M., Ottaviano, G., Tonellotto, N., Perego, R.: Midas: a cloud platform for soa testing as a service. International Journal of High Performance Computing and Networking (2015) **(in press)**
15. Dulz, W., Zhen, F.: MaTeLo—statistical usage testing by annotated sequence diagrams, Markov Chains and TTCN-3. In: Proceedings of the 3rd International Conference on Quality Software (QSIC) (2003)
16. Eclipse Foundation: Papyrus. https://eclipse.org/papyrus/
17. Feliachi, A., Le Guen, H.: Generating transition probabilities for automatic model-based test generation. In: Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST) (2010). doi:10.1109/ICST.2010.26
18. Feller, W.: An introduction to probability theory and its applications. Wiley, Hoboken (1971)
19. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Comput. Netw. **42**(3), 375–403 (2003). doi:10.1016/S1389-1286(03)00249-4
20. GS1: Logistics interoperability model version 1. http://www.gs1.org/lim (2007)
21. Health Level Seven International: Hl7 version 3 standard: Identification service (is), release 1. http://www.hl7.org/implement/standards/product_brief.cfm?product_id=87 (2014)
22. Healthcare Service Specification Project (HSSP): Hssp specifications. https://hssp.wikispaces.com/specs
23. Herbold, S., Bünting, U., Grabowski, J., Waack, S.: Deployable capture/replay supported by internal messages. Adv. Comput. **85**, 327–367 (2012)
24. Herbold, S., Grabowski, J., Waack, S.: A Model for Usage-based Testing of Event-driven Software. In: 3rd International Workshop on Model-Based Verification & Validation From Research to Practice. IEEE Computer Society (2011)
25. Herbold, S., Harms, P.: AutoQUEST—automated quality engineering of event-driven software. In: Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW) (2013). doi:10.1109/ICSTW.2013.23
26. International Software Testing Qualitifications Board (ISTQB): Standard glossary of terms used in Software Testing, Version 2.1 (2010)
27. Kosala, R., Blockeel, H.: Web mining research: a survey. ACM SIGKDD Explor. Newsl. **2**(1), 1–15 (2000). doi:10.1145/360402.360406
28. Le Guen, H., Marie, R., Thelin, T.: Reliability estimation for statistical usage testing using markov chains. In: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE) (2004). doi:10.1109/ISSRE.2004.33
29. Littlewood, B.: A reliability model for systems with Markov structure. J. R. Stat. Soc. Ser. C (Applied Statistics) **24**(2), 172–177 (1975)
30. MIDAS Consortium: Model and Inference Driven Automated testing of Servicesarchitectures (MIDAS). http://www.midas-project.eu (link checked June 2nd, 2014)
31. Motahari-Nezhad, H.R., Saint-Paul, R., Casati, F., Benatallah, B.: Event correlation for process discovery from web service interaction logs. VLDB J. **20**(3), 417–444 (2011). doi:10.1007/s00778-010-0203-9
32. Object Management Group (OMG): Retrieve, locate, and update service (rlus). http://www.omg.org/spec/RLUS/ (2011)
33. Rumbaugh, J., Jacobson, I., Booch, G.: Unified modeling language reference manual, the (2nd edition). Pearson Higher Education, New York (2004)
34. Srivastava, J., Cooley, R., Deshpande, M., Tan, P.N.: Web usage mining: discovery and applications of usage patterns from Web data. ACM SIGKDD Explor. Newsl. **1**(2), 12–23 (2000). doi:10.1145/846183.846188
35. Testing Technologies: Ttworkbench. http://www.testingtech.com/products/ttworkbench.php
36. Tonella, P., Ricca, F.: Dynamic model extraction and statistical analysis of web applications. In: Proceedings of the 4th International Workshop on Web Site Evolution (WSE) (2002)
37. Tonella, P., Ricca, F.: Statistical testing of web applications. J. Softw. Maint. Evol. Res. Pract. **16**(1–2), 103–127 (2004). doi:10.1002/smr.284
38. Tonella, P., Ricca, F.: Dynamic Model extraction and statistical analysis of web applications: Follow-up after 6 years. In: Proceedings of the 10th International Symposium on Web Site Evolution (WSE) (2008)
39. Tonella, P., Tiella, R., Nguyen, C.D.: Interpolated n-grams for model based testing. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 562–572. ACM, New York, NY, USA (2014). doi:10.1145/2568225.2568242
40. Walton, G.H., Poore, J.H., Trammell, C.J.: Statistical testing ofsoftware based on a usage model. Softw. Pract. Ant Exp. **25**(1), 97–108 (1995). doi:10.1002/spe.4380250106
41. Wendland, M.F., Schneider, M., Hoffmann, A.: A model-driven approach to test automation for soa systems. Software Tools for Technology Transer (submitted) (2015)
42. Wesslén, A., Wohlin, C.: Modelling and generation of software usage. In: Proceedings of the 5th International Conference on Software Quality (1995)
43. Whittaker, J.A., Poore, J.H.: Markov analysis of software specifications. ACM Trans. Softw. Eng. Methodol. **2**(1), 93–106 (1993). doi:10.1145/151299.151326
44. Whittaker, J.A., Thomason, M.G.: A Markov chain model for statistical software testing. IEEE Trans. Softw. Eng. **20**(10), 812–824 (1994). doi:10.1109/32.328991
45. Woit, D.M.: Specifying operational profiles for modules. SIGSOFT Softw. Eng. Notes **18**(3), 2–10 (1993). doi:10.1145/174146.154187
46. Woit, D.M.: Conditional-event usage testing. In: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research, CASCON '98, p. 23. IBM Press (1998)
47. World Wide Web Consortium (W3C): Web services addressing (ws-addressing). http://www.w3.org/Submission/ws-addressing/ (2004)
48. Yufang Dan Nicolas Stouls, S.F.C.C.: A Monitoring approach for dynamic service-oriented architecture systems. In: SERVICE COMPUTATION 2012: The Fourth International Conferences on Advanced Service Computing, pp. 20–23. XPS (Xpert Publishing Services) (2012)