The MIDAS Cloud Platform for Testing SOA Applications

Steffen Herbold*, Alberto De Francesco†, Jens Grabowski*, Patrick Harms*, Lom M. Hillah§, Fabrice Kordon¶, Ariele-Paolo Maesano¶, Libero Maesano‖, Claudia Di Napoli‡, Fabio De Rosa‖, Martin A. Schneider**, Nicola Tonellotto†, Marc-Florian Wendland**, Pierre-Henri Wuillemin†

*Institute of Computer Science, University of Göttingen, Germany
Email: {herbold,harms,grabowski}@cs.uni-goettingen.de
†Istituto di Scienza e Tecnologie dell'Informazione - CNR, Pisa, Italy
Email: {alberto.defrancesco,nicola.tonellotto}@isti.cnr.it
‡Istituto di Calcolo e Reti ad Alte Prestazioni - CNR, Naples, Italy
Email: claudia.dinapoli@cnr.it
§CNRS UMR 7606 (LIP6), Université Paris Ouest Nanterre La Défense, F-92001 Nanterre, France
Email: lom-messan.hillah@lip6.fr
¶CNRS UMR 7606 (LIP6), Sorbonne Universités, Université Pierre et Marie Curie, F-75252 Paris, France
Email: {fabrice.kordon,ariele.maesano,pierre-henri.wuillemin}@lip6.fr
‖Simple Engineering, F-75011 Paris, France
Email: {fabio.de-rosa,libero.maesano}@simple-eng.com
**Fraunhofer Institute FOKUS, Berlin, Germany
Email: {martin.schneider,marc-florian.wendland}@fokus.fraunhofer.de

***Abstract*—While Service Oriented Architectures (SOAs) are for many parts deployed online, and today often in a cloud, the testing of the systems still happens mostly locally. In this paper, we want to present the MIDAS Testing as a Service (TaaS), a cloud platform for the testing of SOAs. We focus on the testing of whole SOA orchestrations, a complex task due to the number of potential service interactions and the increasing complexity with each service that joins an orchestration. Since traditional testing does not scale well with such a complex setup, we employ a Model-based Testing (MBT) approach based on the Unified Modeling Language (UML) and the UML Testing Profile (UTP) within MIDAS. Through this, we provide methods for functional testing, security testing, and usage-based testing of service orchestrations. Through harnessing the computational power of the cloud, MIDAS is able to generate and execute complex test scenarios which would be infeasible to run in a local environment.**

I. Introduction

Due to the growing size and complexity of software systems in the last decades, quality assurance of software becomes at the same time more important and more difficult. One direction researches and practioneers have turned towards, which has been gaining traction in recent years [1] is Model-based Testing (MBT). By now, there are multiple tools on the market that support MBT and that are applied in practice. However, the tools themselves are proprietary and often come with high licencing costs. Moreover, the computational resources required for test generation and execution often do not scale well with MBT. This is due to the possibility to derive huge amounts of test cases but the resources for their execution are not available.

There is a growing trend to deploy testing tools on cloud infrastructures, either public or private, in order to provide testing services through the coordinated use of cloud resources according to a pay–per–use business policy [2]. This is due to several reasons. Business applications are becoming more and more dynamic, complex and distributed, so requiring increasingly sophisticated testing techniques and methods to deal with this complexity. In addition, for a growing number of companies it is prohibitive to maintain in–house testing facilities that can mimic real operating environments due to the high cost and technical difficulties, so the demand for new solutions, where the testing process is outsourced and possibly automated, is rapidly increasing. Nevertheless, the migration of legacy test software to the cloud is a complex task and if it is not done correctly, the underlying cloud infrastructure will not be used efficiently.

Within the "Model and Inference Driven - Automated testing of Services architectures" (MIDAS) project, we built a power testing platform directly on the cloud. Therefore, MIDAS comes without upfront licencing costs and scales in terms of computational power. The idea is to provide a Testing as a Service (TaaS) platform on the cloud, where users of MIDAS can come and rent testing resources as they require them. First of all, this shifts the costs from upfront licencing costs to pay-per-use. This is an advantage, especially for smaller companies who may want to try MBT but cannot risk a failed investment. Moreover, MIDAS is elastic due to its cloud nature, which means that the computational resources scale dynamically as required. Because MIDAS is naturally built as a cloud application, it can harness the computational power of the cloud efficiently and execute complex test scenarios which otherwise may be infeasible.

978-1-4799-7125-1/15/$31.00 ©2015 IEEE

The test methods developed as part of the MIDAS project are for testing Service Oriented Architecture (SOA) orchestrations. SOA orchestrations are a good example of the complexity explosion one faces with modern applications. While testing a SOA based service on its own is feasible, once orchestrations of multiple services are considered, the complexity explodes rapidly. Within MIDAS, we counteract this through the elasticity of the cloud and the MBT approach. The MBT approach is based on the MIDAS Domain Specific Language (DSL), a selection of concepts from Unified Modeling Language (UML) and the UML Testing Profile (UTP). However, the MIDAS TaaS is built to be an open and extensible platform for new test methods. With the DSL as foundation, the platform supports test generation and scheduling with methods from functional testing, security testing, and usage-based testing. Moreover, to fully support test automation, the platform contains services for the generation of Testing and Test Control Notation version 3 (TTCN-3) code from test cases described with the MIDAS DSL, as well as TTCN-3 compilation and execution services based on TTworkbench [3].

The remainder of the paper is structured as follows. In Section II, we dicuss the structure of the MIDAS TaaS cloud platform and its general features. This includes how test methods are integrated in the TaaS and made available to MIDAS users. Afterwards, in Section III we present the MIDAS DSL and the methods for functional testing, security testing, and usage-based testing available on the platform. Finally, we conclude this paper and give an outlook on future work in Section IV.

II. The Cloud–based MIDAS Platform

The MIDAS platform is designed and architected according to the SOA computing paradigm, and deployed on a public cloud infrastructure as a software system providing advanced and off–the–shelf testing methods and capabilities, to be paid–per–use in the digital economy era. In fact, from an end–user perspective, the MIDAS platform will be made accessible over the Internet as a multi–tenancy Testing Software as a Service, we refer to as the MIDAS Testing as a Service (MIDAS TaaS).

The cloud service model adopted for the MIDAS cloud deployment is the Infrastructure as a Service (IaaS), mainly because of the possibility it offers to developers to fully control the entire software environment in which their applications are developed [4]. This is a crucial requirement for the MIDAS platform development since it relies on components that are developed and implemented by the MIDAS technical partners in an independent way, and that have to be integrated in a complete application. Furthermore, MIDAS includes legacy commercial software, such as the TTworkbench execution engine [3], that may require specific software support to be included in the MIDAS platform. Finally, since an IaaS cloud relies on virtualization technologies, the portability of the MIDAS platform to different cloud providers is guaranteed. The drawbacks of IaaS cloud solutions may be an additional complexity and effort required to set up and deploy the application.

In accordance to the project requirement of using a public cloud infrastructure, and to the analysis of available cloud providers [5], the Amazon Elastic Computing (EC2) platform

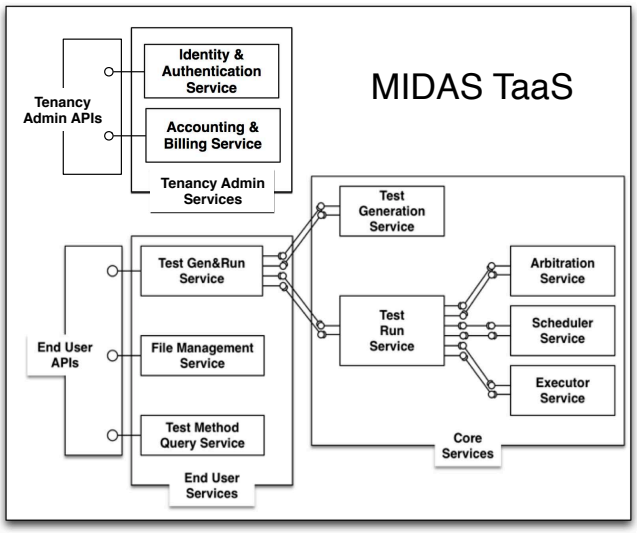

Fig. 1: The MIDAS SOA design

[6] has been adopted. It represents a good candidate solution since it currently offers the best compromise among cost, MIDAS development needs, and elasticity mechanisms. In fact, it provides advanced solutions concerning the possibility to achieve computing elasticity, to tackle data storage reliability, persistency, and fault tolerance.

A. The SOA design of the MIDAS Platform

All functionalities of MIDAS are exposed as services, due to the applied SOA paradigm. The MIDAS services, also referred to as MIDAS components, are asynchronous and stateless Web services accessed through well–defined APIs and communicating with each other. All services are being developed according to their specifications to guarantee their interoperability. In addition, the MIDAS SOA design allows to address the interoperability problem, very crucial in software testing automation, that is achieved through the use of a testing software catalogue (the MIDAS portfolio) of preconfigured and stable testing methods that use shared interfaces.

End users and test method developers are grouped in logical entities called respectively *tenancies* that are separated user computing spaces managed by the corresponding administrators. Conceptually, tenancies represent units of: a) users identification and authentication; b) cloud resources allocation, accounting and billing; c) data and services ownership and access.

The MIDAS services are classified according to their usage, therefore, we allow the management of both the different access policies for their usage, and the different computational resources they need. As reported in Figure 1, the MIDAS services are grouped as follows:

1) **Tenancy Admin services** that include: a) the *Identification & Authentication* service allowing tenancy administrators to manage tenancy end users, and to verify that each member of a tenancy is authenticated before invoking the facilities of that tenancy;

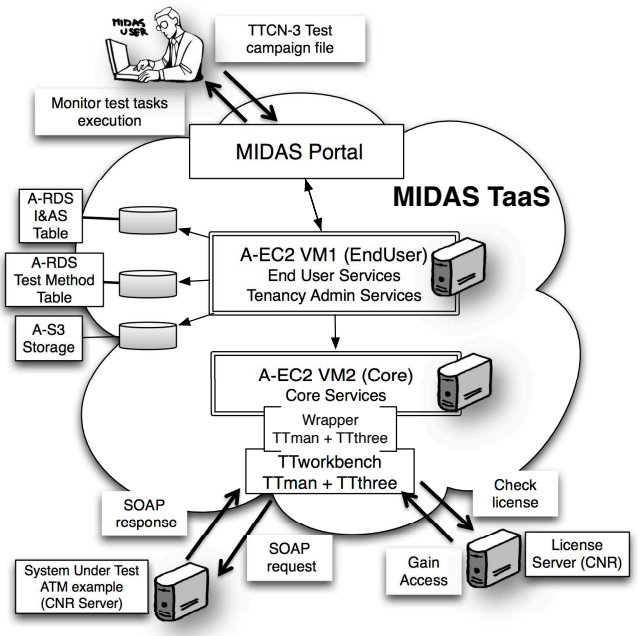

Fig. 2: The MIDAS Prototype on the Cloud

and b) the *Accounting & Billing* service allowing tenancy administrators to monitor the MIDAS cloud resources and services usage of a tenancy, and to get the corresponding updated and consolidated billing information.

2) **End users services** that include: a) the *Test Gen & Run* service allowing end users to asynchronously start the execution of a test task (either a test generation or a test execution task), and to actively poll it to inspect the status and the resulting output of any started test task; b) the *Test Method Query* service, allowing end users to list the test methods currently part of the MIDAS portfolio, and to retrieve the properties of any method in the portfolio; and c) the *File Management* service, allowing end users to access the file system private to the tenancy they belong to, and to perform the usual operations supported by a file system.
3) **Core services** that include: a) the *Test Generation* service, responsible for automatically generating test cases, test scripts and model transformations for testing; and b) the *Test Run* service, that coordinates the run of a specific test cycle, consisting in an optional scheduling phase, a mandatory execution phase, and an optional arbitration phase.

B. The MIDAS Architecture for the Cloud

The deployment of the MIDAS TaaS platform on the cloud takes into account mainly the sandbox and scalability requirements of basic components, as specified in [7]. Each tenancy instance has its own private cloud resources pool that can scale by exploiting the elasticity services of the underlying Amazon cloud infrastructure.

Each tenancy instance is a logical composition of two basic deployment units, (see Figure 2), i.e., two Virtual Machine Images (VMIs), one hosting the Tenancy Administration and End user services (VM1), and the other one hosting the Core services (VM2). The rationale of this choice is due to the different scalability and elasticity requirements of the two groups of services. In fact, most of the workload is expected from the use of the Core Services that host the executor engine, the compiler of TTCN–3 scripts, and the testing components developed by the MIDAS partners. In principle, there is no functional requirement that obliges to have Core services in the same deployment unit. The rationale of grouping these services in a single deployment unit is to allow the entire pool of services working instances and associated resources to be scaled as a whole through Amazon elasticity mechanisms, and to resize the VMs computing resources together with the CPUs, RAM, and network I/O according to the requirements of the MIDAS components.

For what concerns MIDAS storage requirements, MIDAS services need *temporary disk space* for service execution, since each single running service may temporally read/write data whose persistence lasts from the service invocation time to the reply time. A *short–term persistent disk space* is needed to implement custom data sharing among services during single test method generation and run activities. In fact, end user services, like Test Gen & Run, are composite services orchestrating more atomic services, so although all MIDAS services are stateless, component services in an orchestration may communicate exchanging data files through a shared memory disk space. The persistence of these data must last from the invocation time to the reply time of the composite service. Finally, MIDAS must supply a *persistent disk space* for user data including models, test data, test logs, journals and documentation.

The store volumes associated to an Amazon EC2 instance, i.e. the EBS Volumes or Ephemeral Disks, provide a satisfactory solution for the temporary and short–term persistent disk spaces, since they are available until the instance is destroyed. Instead, for the user data persistent space shared among the different users of a tenancy, a suitable solution is Amazon S3. Its data model could also be used to logically partition data among different tenancies in a sandboxed way, if required. The Amazon RDS database solution is adopted for the implementation of the End user, Core and Tenancy admin services requiring to store structured information with frequent accesses. The first prototype of the MIDAS TaaS on the cloud and the mapping of its components to Amazon AWS services are shown in Figure 2.

C. The MIDAS integration strategy

Test method developers are the MIDAS technical partners in charge of developing test methods, and they directly contribute to the development of the platform itself. In fact, the developed test methods are integrated into the MIDAS platform, so becoming part of the complete platform deployed on the cloud, and updated, at each version increment, with the latest release of its components.

To allow the developers to implement and integrate their components and test methods in the MIDAS TPaaS, a virtualization approach was adopted already in the development phase

of the MIDAS platform. The separation between resource provision and operating systems introduced by virtualization technologies is the key enabler for cloud computing, more specifically for IaaS clouds.

The MIDAS test developers have been provided with a seamless, loosely coupled development and integration platform, referred to as the MIDAS Development Environment supporting them in their implementation, debugging and testing activities. The MIDAS Development Environment relies on open–source and stable virtualization technologies, and it is used to develop all the MIDAS components in a consolidated and shared VMI, and to configure and manage all the software packages required for the development of the MIDAS platform components, including third party software dependencies. The shared VMI includes standard hardware architecture, operating system, developer tools, application containers, web servers, and libraries shared among all partners. It is deployed on a local machine by each developer partner, so allowing to locally provide an emulation of the basic MIDAS platform on the cloud. In fact, the MIDAS Development Environment includes all the main building blocks of the MIDAS platform deployed on the cloud. The virtualized MIDAS development environment is deployed on the Amazon cloud infrastructure, and updated every time a new release of its components is available.

The adoption of a shared VMI to develop the components of the MIDAS platform prevents from using cloud resources in the MIDAS development phase, so allowing for a cost–effective strategy to develop and deploy the MIDAS platform on the cloud. In addition, it guarantees the interoperability of the independently developed components since they are released only once they are stable and run in the same shared environment aligned among all partners.

D. The MIDAS Monitoring and Billing services

The MIDAS platform is equipped with an *Accounting and Billing* service responsible for monitoring the usage of both MIDAS services and computing resources on the cloud. The gathered monitoring data is then processed to provide accounting and billing information for the MIDAS tenancies. The service is built on top of the Amazon Account Billing service, and it will be customized according to the adopted MIDAS business model. This is done by associating an Amazon Identity and Access Management (IAM) account to each Tenancy admin created by the MIDAS TaaS administrator to monitor the usage of Amazon EC2 resources (computing resources, elastic load balancer, autoscaling), Amazon RDS resources, Amazon S3 Storage resources, I/O requests, and so on for each tenancy.

In addition, also events at the platform level are monitored within MIDAS, based on interception of all SOAP messages exchanged among MIDAS web services. The basic element of the implementation of the MIDAS monitoring is an application server interceptor, a small software automatically deployed with the MIDAS platform on the cloud in any web application container, which is responsible to automatically process all incoming and outcoming SOAP messages, to extract relevant timestamped information such as user identifier, test method identifier, and target service that can be used by the Accounting and Billing service to perform more elaborated operations.

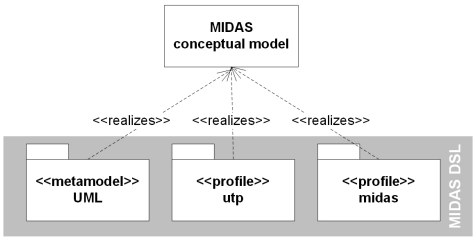

Fig. 3: The MIDAS DSL - Conceptual model and realization

By enabling the Consolidating Billing of Amazon AWS, a monthly report of cloud resources consumption for each tenancy will be stored on Amazon S3 and processed to report all information about cloud resources consumption. Also, the resources consumption of each tenancy user is tracked by keeping the history of user activities and the amount of cpu time spent in each testing task. This information will be used to charge, according to the MIDAS business model, the tenancy users for both cloud resources and MIDAS services usage.

III. SOA Testing with MIDAS

A. The MIDAS DSL

One of the main principles of the MIDAS platform design was to use a single language for input models for each test method based on UML, the so-called MIDAS DSL. The specification and implementation of the MIDAS DSL followed the recommendation of Bran Selic [8] for developing UML profiles. At first, a pure conceptual model was designed. Afterwards, this conceptual model was realized by appropriate modeling languages. The MIDAS DSL is designed as a conglomerate of complementary UML profiles, both standardized and proprietary ones. The standardized parts of the MIDAS DSL are UML [9] and UTP [10]. In such cases where concepts required for a test method developer were not sufficiently addressed by these standards, additional concepts were implemented as a proprietary MIDAS UML profile (see Figure 3).

The MIDAS DSL consists of the conceptual parts test planning, test design, test data, and test scheduling.

Test planning. This part deals with the definition of a test context, or sub-test process (as it is called in ISO 29119). Main concepts of this part are *test plan*, *test requirement*, *test context*, *test level* and *test type*.

Test design. This part offers concepts to describe both *test design models* (artifacts that are used as input for a test generator) and *test cases* (artifacts that represent manually or automatically derived test cases). UML Interactions[1] are used to express test cases. Another important concept of this part are test design directives and test design strategies. They offer the possibility to deposit the chosen test design techniques (such as all transitions, pairwise testing, equivalence class testing etc.) directly within the test model in an abstract manner [11]. With respect to language design, this ability is one of the most important innovations developed within the MIDAS project.

[1]UML Interactions are commonly known as Sequence diagrams.

Test data. Testing is mainly about data exchange between the System Under Test (SUT) and its/their environment. In a test case, data is either used to describe a stimulus to the SUT or an expected response from the SUT. For fundamental data definitions, UML already offers various *ValueSpecifications*. In order to specify efficient and concise test cases, these concepts are not sufficient, in particular for the definition of expected responses. Hence, the MIDAS DSL complements UML and UTP with additional ValueSpecifications realized as Stereotypes. These ValueSpecifications cover *regular expressions*, *collections* and variants thereof (such as sets, subsets, supersets etc.), and *enumerated values*.

Test scheduling. Test scheduling enables the tester to order the respective test cases for execution according to a certain test strategy. A risk-aware scheduling could be such a test strategy. Test schedules are expressed as UML Activities within a test context.

The MIDAS platform also offers a dedicated MIDAS test model to executable TTCN-3 transformation. The generation of executable TTCN-3 test scripts from UTP-based models has been previously discussed ([12], [13]). We contributed to this work by extending the transformation with the newly developed concepts of the MIDAS profile. This guaranteed the applicability and feasibility of the concepts.

The experiences made with the MIDAS DSL, its benefits, as well as its drawbacks provided valuable feedback for the currently undergoing efforts towards a major revision of the UTP, i.e., UTP 2. Many parts of the MIDAS conceptual model and the realizing UML profile implementations have been contributed to the development of UTP 2. The most important findings from developing and applying the MIDAS DSL were the test design directives and test design strategies, as well as the additional ValueSpecifications.

B. Functional Testing

The MIDAS platform aims at the full automation of testing methods. Service functional conformance test automation is by definition model-based: what can be automated is the test of the compliance of the service architecture under test (SAUT) with its formal model. The idea behind extreme automation is that the only manual task for the developer is the production of the SAUT models that, once built reduces the marginal cost of human effort to zero. The service functional testing activity is decomposable in the tasks listed below organised in two main test cycles: (i) *test generation cycle* (test case production, test oracle production) and (ii) *test run cycle* (test execution/arbitration, test reporting, test scheduling).

1) Test generation cycle: having the strongest impact on the effectiveness and efficiency of software testing, test case generation has been implemented by a wide range of intensive techniques that aim at automating the approach. Notable techniques are model-based, model-checking-based, random, and search-based testing [14]. For SOA, formal techniques have the strongest impact on functional testing. We specifically combine symbolic execution and verification techniques based on Petri nets for oracle and test case generation. To generate test oracles, we mainly use inputs describing the external expected behaviour of each participant in the SAUT, the system architecture service composition (SCA) [15], and the services interfaces (in WSDL). State Charts XML (SCXML) are used to specify the behavioural models [16].

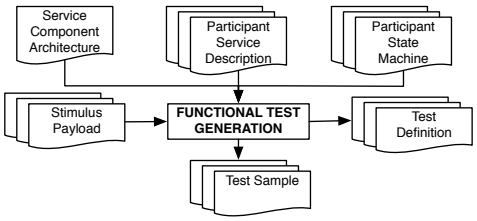

Fig. 4: Inputs/outputs of functional tests generation

Figure 4 shows the inputs and outputs of the functional test generation method, from a user perspective. All inputs and outputs are XML documents, which allows to cross reference elements from one document to another, and enables consistency checking. A user of this method will have to provide, apart from the service description (WSDL), the service component architecture (SCA) and each participant's protocol state machine (SCXML). At least a payload (i.e. SOAP message as input data) is requested for each stimulus, to help the test generation method in inferring further input payloads for that stimulus. A test definition specifies the execution path that a test case yields, while a test sample is the associated data flow (exchanged SOAP messages along the execution path), given the stimulus payload.

Different strategies are combined to generate relevant data to build the oracles for the test cases. Explored strategies are random, boundary values, and domain partitioning by analysing transfer functions (business rules in the SCXML) manipulating values across the exchanged SOAP messages. In addition, hints may be provided for test case generation, such as specific ranges of values and specific types of message.

2) Test Scheduling: Test scheduling gives a specific order of running to the test cases of a test suite. There is a distinction between (i) static and (ii) dynamic scheduling. Static scheduling is batch scheduling, which is also known as prioritisation. Dynamic scheduling is the choice at run time of the next test case to run. Dynamic scheduling requires the action of an artificial agent that is able to decide at each Test run cycle the test case to run on the basis of a decision that takes into account the context in which the cycle is situated and the history of the past cycles. In principle, the scheduler should decide the next test case to run on the basis of its fitness to different criteria (for instance the fault-exposing potential [17]). With dynamic scheduling the fitness of each not-yet-performed test case can change at each cycle on the basis of the evolving test session context and the past test verdicts. Automated test scheduling requires the automation of methods for strategic reasoning and troubleshooting. If the automated test execution/arbitration is available, the complete automation of the dynamically scheduled test run cycle is possible.

a) Probabilistic Modelling: The probability theory has proven to be one of the most promising frameworks for representing uncertainty within a decision framework thanks to its ability of modelling a complex reality with maximum accuracy and minimum number of parameters. Our claim is

that probabilistic inference can drive dynamic test scheduling that improves failure seeking and troubleshooting. The Scheduler decision module follows a (model-based) probabilistic test scheduling approach. In order to initialise and configure its embedded inference engine, the Scheduler builds a probabilistic model (Bayesian Network, BN) of the SAUT by using the functional hierarchical model of the SAUT, the model of the test scenarios and the Test Suite data set.

b) Functionnal SOA Testing Inference: The scheduler compiles the probabilistic model of the SAUT into an Arithmetic Circuit (AC, [18]) by using an original compilation algorithm [19]. The inference engine on the AC is an internal module of the Scheduler. To each execute/arbitrate cycle (with test samples as inputs and test verdicts as outputs) corresponds a schedule inference cycle (with test verdicts as inputs and test samples as outputs). The Scheduler manages its internal inference engine by setting: (i) prior probabilities on its top variables at the initialisation and (ii) assumptions/beliefs/observations (evidence realisations) on the other variables potentially at any inference cycle. The inference permits to calculate $P(X|e)$ where X is an unobserved variable and e is the evidence of the actual state of the scheduler and then the fitness of each non-executed test case. Different policies are proposed to exploit those probabilities and influence the decision of the next test case ([20], [21]).

3) Evidence-driven Test Case Generation: Evidence-driven test case generation shall be intended in the context of this research as the arrangement and timetabling of the tasks described above. In particular, evidence-driven test case generation organises the test generation cycle, the test run cycle and the relationship between them. In principle, the test generation cycle and the test run cycle can be separated and managed independently. The test generation cycles produce collections of test samples (test suites) that are stored for future use. These test suites are inputs of the test run cycle that is performed asynchronously. Advanced test scheduling can supply inputs to test planning, by indicating, on the basis of the verdicts of past test runs, some specifications about the production of new test cases. The Scheduler should supply directives and data to the test case production task, for example by focusing the test generation activity on the coverage of specific regions of the service component architecture or on specific type of messages or, on the contrary, by conducting a breadth-first search for failures.

C. Security Testing

Security Testing means to find security relevant weaknesses in the implementation of a system or a web service. Such weaknesses may be exploited by an attacker, e.g. in order to crash it, get access to prohibited data or functionality, or manipulate it, i.e. to perform actions that may affect its confidentiality, integrity, or availability (CIA). With respect to the daily attacks on systems by foreign governments and intelligence agencies as well as hacker groups, it is inevitable to find as many weaknesses as possible before releasing a system or web service. As web services are naturally exposed to networks and thus, they are easier to attack than other systems. As an integral part, the MIDAS platform also provides services to perform security testing of web services. For this purpose, two different approaches of fuzzing techniques are employed, i.e. data fuzzing and behavioral fuzzing. Additionally, a first approach of the combination of both techniques is implemented.

1) Data Fuzzing: Data fuzzing is an established technique where invalid or unexpected input data is injected to the interface of a system. A secure way of a system to handle such data may be rejecting it and responding with an error message or sanitizing it by replacing or escaping malicious parts of it. However, to do so, the system has to detect that the received data is invalid. This task is performed by input validation mechanisms. However, if these mechanisms are faulty or even missing, invalid input data is able to pass the interface and gets processed by the business logic. Achieving this allows the modification of the behavior of a system in a malicious way as described above. Data fuzzing aims at generating such invalid input data that is able to reveal faulty input validation mechanisms.

As the possible input space for invalid input data generated by data fuzzing techniques is huge, we avoid that the model contains all the fuzz test data. To achieve this, we employ a mechanism called *test strategies*. Test strategies are a way to determine how test cases or test data shall be derived from a given model. We developed a UML profile that contains test strategies specific for data fuzzing. A test strategy specifies how fuzz test data shall be generated for a given message parameter using its type and possible valid data. By this approach, the model is kept small and contains the minimal set of test cases that are abstract in that way that they contain only valid but no invalid, i.e. fuzzed data. Actual fuzz test data are generated at test execution time.

These test strategies are implemented by the fuzzing heuristics the data fuzzing library Fuzzino[2] is providing. Fuzzino is an Open Source test data generator for fuzzing that supports test data generation for String and number data types based on fuzzing heuristics from the popular Open Source fuzzers Peach and Sulley. We extended Fuzzino in order to support and fuzz additionally data structures important for web services.

2) Behavioral Fuzzing: Another kind of fuzzing is called behavioral fuzzing [22]. In contrast to data fuzzing, invalid message sequences are generated from valid ones. Different behavioral fuzzing operators are modifying functional test cases in order to transform them to behavioral fuzzed test cases that generate invalid message sequences. Doing so, it is possible to find security-relevant weaknesses for instance in the authentication and authorization mechanisms. In addition, we are also investigating the combination of both, data and behavioral fuzzing. This means to send invalid messages as well as invalid data such that weaknesses can be found more efficiently than employing merely one technique and in order to find weaknesses that a single technique would not find. Figure 5 depicts the principle of test generation for behavioral fuzzing. A functional test case (on the left) is modified by a behavioral fuzzing operator *RepeatMessage* such that the HTTP get request contains two *host* messages instead one. Since only one host message is allowed, the resulting message sequence is invalid.

We use functional test cases in form of UML sequence diagrams as starting point for our fuzz test case generation.

[2]http://github.com/fraunhoferfokus/Fuzzino

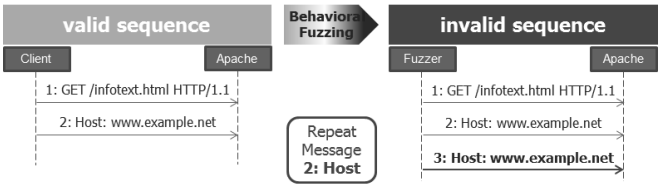

Fig. 5: Example of the generation of a behavioral fuzz test case.

This is reasonable because the SUT should be functionally tested before security testing is performed. The reason is that as long as functional bugs are being fixed, new weaknesses may be induced as well as existing ones may be removed. Therefore, security testing should start when functional testing has been finished. On the other hand, security testing can benefit from functional testing by reusing the functional test cases as starting point for the different fuzzing techniques aforementioned.

As described above, the test case space is that huge that executing all test cases is usually impossible. This is true for data fuzzing as well for behavioral fuzzing and even more for the combination of both techniques. In order to cope with this challenge, we developed a metric for the complexity of the negative input space. It is based on works from Cataldo et al. [23] as well as Bandi et al. [24] who investigated a correlation between interface complexity and error proneness of its implementation. We make use of this correlation by adapting it for data fuzzing where the complexity of this space—e.g. number of boundaries of valid value ranges, (conditional) dependencies between parameters—seems to be relevant for the error proneness of the corresponding input validation mechanisms. Therefore, we calculate a score based on the negative input space complexity to schedule the data fuzz test cases aiming at executing test cases with a higher probability to find a weakness.

D. Usage-based Testing

The usage-based testing part of the project aims at bringing existing techniques for usage-based testing (e.g., [25], [26]) to a level, where they are ready to be applied in the industry. Usage-based testing is based on the idea to optimize the user-experienced quality [27] and, therefore, it generates and/or selects tests that cover highly used parts of the software. In general, usage-based testing consists of three steps:

1) obtain a usage journal;
2) create a usage profile; and
3) select/generate tests.

With MIDAS, we provide a complete tool set to automate all three steps. The tooling is based on the AutoQUEST platform [28] for observation-based quality assurance. MIDAS reuses existing AutoQUEST components and adds additional components to the framework in order to harness the functionalities for usage-based testing provided by AutoQUEST.

To obtain the usage journal, we record each SOAP message exchange between services of a SOA. This is done using Hypertext Transfer Protocol (HTTP) proxies located in front of any service. A proxy receives any request sent to a specific service, forwards it to the service, receives the services response, stores the request and the response in eXtensible Markup Language (XML) format into a log file, and sends the service response to the client. Through this, we get a usage journal of an individual service being made up of XML encoded message exchanges in a log file. The journals of many services can be combined to a usage journal of the whole SOA.

The gathered usage data can then be handled by MIDAS services to create a usage profile. The usage profiles are Markov models that describe the probability of the next SOAP message. Figure 6 shows a small example of a usage profile where a user calls operations from two services called *ixsqm* and *rlus*. MIDAS provides a service for the automated inference of a usage profile based on the usage journal. The inferred usage profile is stored on the platform, and it can then be exploited by usage-based testing techniques.

MIDAS offers multiple ways to exploit the usage profiles. The most important one is the automated generation of tests through random walks. This way, MIDAS users are able to create a test suite, where the test cases mimic the behavior of the service's users. The generated tests are in form of UML interactions and compliant to the MIDAS DSL (see Section III-A). In combination with the services for TTCN-3 generation, compilation and execution, an automated testing approach from usage observation via over usage profile inference, test suite generation, and test execution is possible.

Moreover, MIDAS provides a usage-based scheduling service that is also based on usage profiles. Given a test suite described in the MIDAS DSL (i.e., as UML interactions), MIDAS calculates a *usage score* for each test case, where the usage score reflects the likelihood that the scenario executed by the test case would be executed by the user. The scheduling service then creates a schedule for the test cases where the test case with the highest likelihood is executed first, then the second highest likelihood, etc. This ensures that the tests that affect the user experienced quality are executed first, which is important if not all tests can be executed, e.g., due to a lack of time or available resources.

IV. Conclusion

In this paper, we presented the MIDAS TaaS platform for the MBT of SOA orchestrations. The MBT is based on the MIDAS DSL, one of the key drivers of the standardization of the UTP version 2.0 at the Object Mangement Group (OMG) and the International Organization for Standardization (ISO). The platform is open and features a flexible functionality for the development of new test methods. Currently, functional testing, security testing and usage-based testing are supported. Finally, since the MIDAS platform is provided as a Testing as a Service on the cloud, it allows end users to use it by renting testing resources as they require them according to a pay-per-use business policy. This is advantageous mainly for small and medium enterprises for which the cost of in-house testing may become prohibitive from both an economic and technical point of view. In addition, the design of the MIDAS platform as a SOA cloud-based platform, allowed to effectively exploit the features of the underlying cloud infrastructure, that is usually more difficult when simply migrating already existing applications to the cloud.

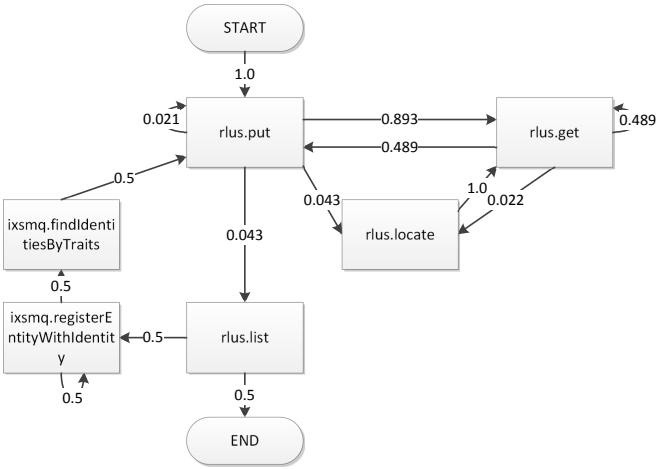

Fig. 6: Example of a usage profile. The first part of each node is the service that is called, the second part the operation of the service, e.g., *rlus.put* is a call of the *put* operation of the *rlus* service. *START* and *END* are virtual operations that signify the start/end of a user session. The edges denote the probability of operation calls given the last operation, e.g., the operation *rlus.list* is called with a probability of 0.043 after the *rlus.put* operation.

In the future, the test methods will be further extended, e.g., by expanding the functional testing to so-called evidence-based testing, where the tests are dynamically scheduled and created based on the test outcomes. Furthermore, the combination of fuzzing and usage-based testing is a promissing approach for security testing that will be investigated. Finally, through cooperations, e.g., with the FITTEST project [29], more test methods shall be brought to the platform.

ACKNOWLEDGMENT

This work was done in the context of the MIDAS European project (project number 318786).

REFERENCES

[1] R. V. Binder, A. Kramer, and B. Legeard, "2014 Model-based Testing User Survey: Results," http://model-based-testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf, 2014.

[2] M. Beisser, M. Pruksch, and S. Limmer, "Test@cloud - mbt and cloud-testing - a powerful combination," in *2nd ETSI User Conference on Advances in Automated Testing (UCAAT)*, 2014.

[3] Testing Technologies, "TTworkbench - The Reliable Test Automation Platform," http://www.testingtech.com/products/ttworkbench.php.

[4] J. Varia, "Architecting for the cloud: Best practices," AWS Whitepaper, Tech. Rep., 2010.

[5] The MIDAS consortium, "Specification and design of the basic midas platform as a service on the cloud. midas deliverable d6.2." MIDAS Deliverable D6.2.

[6] Amazon, "Aws - amazon elastic compute cloud (ec2)," http://aws.amazon.com.

[7] The MIDAS consortium, "Architecture and specifications of the midas framework and platform," MIDAS Deliverable D2.2.

[8] B. Selic, "A systematic approach to domain-specific language design using UML," in *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece*, 2007, pp. 2–9. [Online]. Available: http://dx.doi.org/10.1109/ISORC.2007.10

[9] Object Management Group (OMG), "Unified modeling language (uml)," http://www.omg.org/spec/UML/.

[10] ——, "Uml testing profile, version 1.2," http://www.omg.org/spec/UTP/1.2/.

[11] M.-F. Wendland, "Abstractions on test design techniques," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, Sept 2014, pp. 1575–1584.

[12] J. Zander, Z. Dai, I. Schieferdecker, and G. Din, "From u2tp models to executable tests with ttcn-3 - an approach to model driven testing -," in *Testing of Communicating Systems*, ser. Lecture Notes in Computer Science, F. Khendek and R. Dssouli, Eds. Springer Berlin Heidelberg, 2005, vol. 3502, pp. 289–303. [Online]. Available: http://dx.doi.org/10.1007/11430230_20

[13] M.-F. Wendland, M. Kranz, C. Hein, T. Ritter, and A. García Flaquer, "Model-based testing in legacy software modernization: An experience report," in *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, ser. JAMAICA 2013. New York, NY, USA: ACM, 2013, pp. 35–40. [Online]. Available: http://doi.acm.org/10.1145/2489280.2489291

[14] S. Anand, E. K. Burke, T. Y. Chen, J. A. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2013.02.061

[15] OASIS, *Service Component Architecture (SCA)*, http://www.oasis-opencsa.org/sca, 2011.

[16] W3C, *State Chart XML (SCXML): State Machine Notation for Control Abstraction*, http://www.w3.org/TR/scxml/, May 2014.

[17] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/32.988497

[18] A. Darwiche, "A differential approach to inference in bayesian networks," *J. ACM*, vol. 50, no. 3, pp. 280–305, May 2003. [Online]. Available: http://doi.acm.org/10.1145/765568.765570

[19] The MIDAS consortium, "Probabilistic inference engine for test planning and scheduling," MIDAS Deliverable D5.3.

[20] D. Heckerman, J. S. Breese, and K. Rommelse, "Decision-theoretic troubleshooting," *Communications of the ACM*, vol. 38, pp. 49–57, 1995.

[21] W. Rödder, I. R. Gartner, and S. Rudolph, "An entropy-driven expert system shell applied to portfolio selection," *Expert Syst. Appl.*, vol. 37, no. 12, pp. 7509–7520, Dec. 2010. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2010.04.095

[22] M. Schneider, J. Gromann, N. Tcholtchev, I. Schieferdecker, and A. Pietschker, "Behavioral fuzzing operators for uml sequence diagrams," vol. 7744, pp. 88–104, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36757-1_6

[23] M. Cataldo, C. R. B. D. Souza, D. L. Bentolila, T. C. Mir, and S. Nambiar, "The impact of interface complexity on failures: an empirical analysis and implications for tool design," Carnegie Mellon University, Techn. report CMU-ISR-10-100, 2010.

[24] R. Bandi, V. Vaishnavi, and D. Turk, "Predicting maintenance performance using object-oriented design complexity metrics," *Software Engineering, IEEE Transactions on*, vol. 29, no. 1, pp. 77–87, Jan 2003.

[25] P. Tonella and F. Ricca, "Statistical testing of web applications," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 1-2, pp. 103–127, 2004.

[26] S. Herbold, "Usage-based Testing of Event-driven Software," Ph.D. dissertation, Dissertation, Universität Göttingen (electronically published on http://webdoc.sub.gwdg.de/diss/2012/herbold/), June 2012.

[27] International Software Testing Qualitifications Board (ISTQB), "Standard glossary of terms used in Software Testing, Version 2.1," 4 2010.

[28] S. Herbold and P. Harms, "AutoQUEST – Automated Quality Engineering of Event-Driven Software," in *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, March 2013.

[29] "Fittest," http://crest.cs.ucl.ac.uk/fittest/.