A Generic Testing Framework for the Internet of Services

Senol Arikan, Aneta Kabzeva, Joachim Götze, Paul Müller
ICSY - Integrated Communication Systems
TU Kaiserslautern
Germany
{arikan, kabzeva, j_goetze, pmueller}@informatik.uni-kl.de

***Abstract*— A widespread approach of the design and development of heterogeneous distributed software systems is the use of an interacting group of services. This approach uses the concepts of Service-oriented architectures to realize a dynamic adaptive communication system among service provider, service consumer and service broker. Today, the Internet, as the largest heterogeneous distributed software system, uses the ideas of service orientation more and more, thus it is extended to the Internet of Services. In the Internet of Services, autonomous services can be deployed by different service providers on service platforms; thereby they are available via the Internet for a large number of service consumers. Service providers can change their service implementation at any time without notifying the service's consumers; therefore, no guarantees can be made about the adherence of these services to the specification the consumers expect. In order to ensure the compliance of the services to their specifications and provide a certain level of quality assurance for service consumers and platform providers, a service platform needs to provide comprehensive testing mechanisms which support the quality needs of all actors on the platform. In this paper, we propose a generic testing framework which can be used during design and run-time for the automated verification of distributed services.**

Keywords- SOA; Internet of Service; run-time testing; black-box testing; verification

I. INTRODUCTION

The quality of large software systems is one of the most important goals of software development. Balzert [8] defines software quality as the sum of the following characteristics: functionality, usability, reliability, performance, maintainability. All these characteristics define the degree to which a software product fulfills its functional and non-functional requirements.

One fundamental prerequisite for software quality is the software's robustness to possible faults. This can, for example, be achieved through early identification and correction of failures [12]. In general, a failure in a system means that a wanted behavior is not achievable (i.e., a behavior which does not conform to the requirements has occurred. In order to detect such a system behavior, different testing strategies can be performed. A fundamental classification divides testing strategies into black-box and white-box testing.

The complexity of software testing reaches a new level with the need to test heterogeneous distributed software systems. A widespread approach for the design and development of heterogeneous distributed software systems is the use of an interacting group of services. This concept is based on the architectural principle "separation of concerns", which focuses on one simple, well-known idea: a large problem is more effectively solved if it can be broken down into a set of smaller problems [11]. Service-oriented architectures (SOAs) solve complex concerns using service orientation. It is important to have loosely coupled services so that failures or changes in one service do not cause failures in other services. Service-oriented architectures realize a dynamic-adaptive communication system among service providers, service consumers, and service brokers. In the Internet of Services (IoS) [9], loosely coupled, reusable, autonomous services can be deployed by different service providers on service platforms. These services are distributed by the platform provider and, as such, are reachable over the Internet by a huge number of service consumers. Since services are offered by different service providers, no guarantees can be made about the functional adherence of these services' implementations to their previously defined specifications. Providers can change their service implementations and introduce new bugs at any time without notifying the service consumers [14]. Additionally, platform providers cannot be sure about the performance of their services. In order to test the services' compliance to their specifications and provide a certain level of quality assurance for service consumers and platform providers, service platforms need to provide comprehensive testing mechanisms which support the quality needs of all actors on the platform.

Unfortunately, platform providers typically do not have access to the services' source code as a standard service design principle, abstraction, is that no information about the internal realization of a service has to be published for other actors of the distributed system. Hence, the implementation of a service is unknown for a service consumer and platform provider. This fundamental characteristic along with the need for service providers and platform providers to ensure the quality of their services forces service testing to focus on black-box testing approaches in the Internet of Services. The service tester, independent of his role on the platform, should be able to invoke a service with a specific test case to check the response of the service. If the service output doesn’t conform to the expected value, then the service does not meet the expected quality for that test case.

To ensure the conformity of a service implementation with its service specification during its complete lifetime, a service must be tested not only during development but also at run-time. We propose a generic testing framework which can be used for the automated verification of services during their complete life cycle. The proposed solution is based on black-box testing. An evaluation of the concept is provided by a prototype implementation of the framework in an existing SOA-based infrastructure. The framework takes as input user-defined valid test cases and generates test clients, which are executed to try out the desired service functionalities for suitable parameters. Analysis of the test results and notification of affected actors are also important requirements to address the challenges of a testing framework for the Internet of Services.

In order to address the quality requirements of all relevant stakeholders at any time in the service lifetime, the framework supports stress tests, scalability tests and parallel tests. To be able to support the testing of the potentially large number of resources offered on a service platform and the dynamic number of testing requests coming from consumers, the proposed solution considers asynchronous and synchronous communication models.

The paper is structured as follows. In the next section, we present some related work. In Section III, we explain the basics needed for understanding the work. Section IV describes the challenges for testing in the context of Internet of Services in more detail. Section V presents the architecture of the testing framework proposed as a solution addressing these challenges followed by an implementation approach. Section VI concludes the paper and discusses identified future work.

II. RELATED WORK

In this section, we give an overview regarding the different solution proposals from other researchers for testing service-based distributed systems. The approach of Looker, Munro and Xu [1] concentrates on measurement techniques to test the robustness of Web services using *network level fault injection* to manipulate the expected parameters of a service at run-time. This approach has the disadvantage of requiring the service’s source code in order to make required modifications. A service tester who does not have access to the service’s implementation cannot use this approach to test a service. Our framework uses a black-box technique, thus it enables testing of a service for each stakeholder involved in the life cycle of a service. Frantzen, Tretmans, and Vries [2] apply a *model-based testing technique* to experiment with a Web service, which aims at either finding faults or gaining confidence in the service. Model-based techniques have been developed for reactive systems. In order to apply techniques for MBT (Model-Based Testing) of reactive systems in SOA-based systems, some additional requirements must be satisfied. These additional requirements can increase the complexity of the realization of the proposed approach.

Most of the solutions for testing of service-based distributed software systems have experimented with SOAP-RPC based Web Services. Chakrabarti and Kumar [3] have developed a black-box approach for testing RESTful Web Services which uses a test specification language for better automation in test execution. This approach is limited to testing RESTfull services.

To the best of our knowledge, the only works which address issues close to ours are [4] and [5]. Martin, Basu, Xie [4] presents a unit testing framework for Web services based on JUnit. This framework uses the test generation tool JCrasher in order to generate corresponding JUnit tests. WS-TAXI [5] is a WSDL-based testing tool for Web Services, which is obtained by soapUI [6], an industrial testing tool, and TAXI [7], which automatically generates XML-based test cases from a corresponding XML schema. This framework is based on the idea of automatic generation of SOAP envelopes by using data instances from WSDL descriptions, which are used for service invocation. Our framework does not use any external tools for the generation of test cases. With only minimal amount of input data, which are given by service testers in XML-format, and with use of WSDL descriptions, the test clients will be automatically generated and executed at run time. We concentrate on the quality of the testing process and developing an efficient and dependable framework, which is highly performant and supports service testers during the whole testing process. Finally, the testers will be notified about analyzed test results. In next section, we go into details and present some basic terms and strategies for testing.

III. TESTING BASICS

In this section we will first define some of the terms that are commonly used when discussing testing. Then we will discuss the details of the two basic testing approaches - white-box and black-box testing, which were mentioned above.

A. Error, Fault, and Failure

There is considerable confusion regarding definitions of error, fault, and failure in the literature. We use the definitions from Jalote for these terms [14]. The term error is used to refer to any activity of a programmer which results in software containing a defect or fault. A fault is a condition that causes a system to fail in performing its required function. A failure is the inability of a system or component to perform a required function according to its specification.

B. Validation and Verification

In general, there are two important evaluation methods to check software against its specification: *verification* and *validation*. As defined by the IEEE [16], *verification* is a process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase; *validation* is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. Using these definitions, *validation* is a process to demonstrate that the software implements each of the functional requirements correctly and completely; *verification* is the process to ensure the software product of a given phase fully implements the inputs to that phase. The framework proposed in this work can be used in any service life cycle stage and therefore supports both the verification and validation of services.

C. *Software testing approaches*

Software testing approaches traditionally divide into black-box and white-box testing.

White-box testing approaches consider the internal data flow and logic of the system under test. This approach is also known as glass-box testing or structural testing. The internal working of the software is visible for the tester. Because the implementation of the software product is known, white-box-testing enables a tester to design test cases that exercise the independent paths within a module or unit, check logical decisions on both their true and false side, execute loops at their boundaries and test the validation of the internal data structure [13]. White-box testing gives a tester a certain amount of control during the testing process. If a fault is detected, the tester knows which lines of code to look at based on the corresponding test case. Because of this control, defining and removing faults in the tested object is more economical and successful than with other testing approaches. The internal data and logic flow of a service is known only to the service developer in the context of Internet of Services. A service provider is not necessarily a service developer and therefore will have no knowledge of the details needed for specifying white-box test cases. This is also the case for consumers of these services.

Black-box testing approaches, also called behavioral testing, consider the tested system as a whole and ignore internal structure details. In contrast to white-box testing, black-box testing is usually used when the implementation of the software is not known to the tester. Black-box testing uses the functional requirements and specifications of the software to define test cases that should fully exercise all the functional requirements [14]. These resources are available for platform actors in IoS: each service has to provide a specification of its functionality which an interested actor can use to define a desirable test case. After generation of the test cases from a specification, some valid and invalid input data are provided for test execution, and then the testing method calls the corresponding software to verify whether the test results are compatible with the expected outputs. Black-box testing terminates when all test cases are executed. According to Pressman [13], black-box testing techniques are applied to find errors in the following categories: incorrect or missing functions, interface errors, errors in data structures or external database access, behavior or performance errors, and initialization and termination errors. An important disadvantage of black-box testing is that it does not help in finding the reason of the failure. Testing the code (implementation) quality is not possible.

Another well known problem of black-box testing is the selection of test cases. In order to deal with this problem, some black-box strategies were defined; they differ according to test case selection criteria [12]. The first strategy is Equivalence Class Partitioning. The idea behind this strategy is to reduce the complexity of selecting test cases by dividing the set of all possible inputs for a function into a set of equivalence classes so that if any test in an equivalence class succeeds, then every test in that class will succeed [14]. Experience shows that faults often occur on the boundaries of equivalent classes [12]. Boundary Value Analysis is based on the experiences gained through Equivalence Class Partitioning and selects test cases which lie on the boundaries of equivalence classes. The goal is to reach a maximal number of tests with as few test cases as possible. Thus the complexity of the testing process can be reduced. Another strategy for black-box testing is Cause-Effect Graphing [14]. This strategy attempts to combine inputs from different input classes through the use of the Boolean operators “and”, “or”, and “not” in order to exercise some special test cases. The disadvantage of this approach is that it can result in a large number of test cases, many of which will not be useful for detecting new faults.

The prototype implementation of the proposed testing framework uses a randomized algorithm which gets random service relevant test cases from a database and executes them. As part of our future work we will specify algorithms based on the Boundary Value Analysis strategy.

In conclusion, black-box testing is not an alternative to white-box techniques. It can be considered as a complementary approach that returns a different class of errors than white-box testing [13]. It means we can also combine both strategies to test a software system, if the corresponding requirements (i.e., availability of source code) can be met.

IV. INTERNET OF SERVICES

In this section we will first introduce concept Internet of Services and then we will define some challenges which should be considered by the testing framework.

A. *Introduction*

With the adoption of the SOA paradigm for the design of distributed business processes [11] and the introduction of Cloud infrastructures that allow on-demand delivery of IT resources [21], the offering, discovery, and usage of technical services over the Internet is not a vision anymore, but a fact. Considering services as tradable goods over the Internet is the main concern of the Internet of Services community [9]. Yet, the opportunity to offer services reachable by a wide range of potential customers on platforms provided by third parties gives rise to some new roles. Each of these roles has its own requirements for the quality of service provided by the services offered on such a platform. Thus, a generic testing framework, used as a major quality control mechanism, should be able to address some of the new role-specific requirements.

Besides the typical software engineering roles of provider and consumer, the SOA paradigm introduces the role of a service broker, which serves as an intermediary between the provider and consumer [11]. In the Internet of Services, the three classical SOA roles are considered insufficient [18][19]. Additionally, the platform provider and service developer roles have to be considered.

In the Internet of Services, the service broker role is taken by the platform controlling the life-cycle of the services it offers [18]. A platform does not only manage a catalogue of offered services and their descriptions, but has also takes care of platform-wide security and quality standards. Stakeholders carrying these responsibilities in the Internet of Services are referred to as platform providers. The platform provider is responsible for providing qualitative infrastructure whose customers are the service providers.

Compared to traditional mainframe applications where the operating organization is normally also the supplier of the software, the Internet of Services makes a distinction between service developer and service provider [19]. Nevertheless, these two roles are not mutually exclusive. A service developer concentrates only on writing the executable code behind a service. This is the only role that has knowledge of the internal logic and data flow within the code. The rest of the stakeholders only have access to the service through its interface description. A service developer has to guarantee the quality of the code only against the service provider. Service providers are responsible for the deployment of services on a platform and the specification of service level agreements (SLAs) [22]. They provide services that offer some value to the service consumers and use the resources of the platform to communicate with their customers. A service provider is accountable for granting the quality of service specified in the SLAs and for compensations in case of violations.

The service consumer uses the platform to find one or more services which can fulfill his needs. The product which is of interest for the service consumer is the real-world effect provided by the functionality of a service. Once a suitable service is identified, a contract has to be negotiated between the consumer and the provider of the service [20]. Since a selected service will probably be integrated in the consumer's operational environment, the consumer has to be given the possibility to test if the service quality still fits the requirements of his own environment at any time.

In addition to the extended number of roles in the Internet of Services, the dynamic organization of service-based distributed systems also introduces some challenges to the execution of tests in such an environment. The changing number of stakeholders acting on a Cloud platform may lead to a large number of test cases that should be covered by the platform testing framework. Since some of the stakeholder roles, like the service developer and the service provider, are interested in design-time tests, and all roles have to be able to check the compliance of the resources to the negotiated contracts at any time, tests should be executable at both design and run time. Some test cases would be executable on demand (i.e., after changes or failure corrections). Others, like tests checking the compliance with SLA terms, should be executable on a regular basis. When quality violations are discovered in the testing process, the testing framework should be able to send the right information to all affected stakeholders; which requires the integration of a notification mechanism within the framework.

In the following section, we present a list of challenges for testing SOA-based distributed software systems from an IoS perspective.

B. *List of challenges*

Considering the relationships, responsibilities, and organization of a service platform in an IoS environment, we identified the following challenges that should be addressed by a testing framework:

- *Large number of test cases*: the number of stakeholders interacting on the platform is variable. Any number of users can join the platform; any number of services can be deployed on the platform. As a consequence, the platform must be able to provide for the execution of the continuously growing number of test cases by making scaling the framework a core part of the implementation.
- *Lack of knowledge on service structure:* for all stakeholders except for service developers, services are only known through their interfaces, the service implementation and structure are intentionally hidden. This makes white-box testing impossible and forces black-box testing.
- *Service life-cycle:* once deployed a service should be always available and cannot be taken offline for maintenance. Thus it is important to provide testing support during service development as well as during service run-time.
- *Different responsibilities*: depending on their role on the platform, different stakeholders have different responsibilities, as explained in the previous section. A testing framework for the Internet of Services should be able to support different kinds of testing in order to cover all role-specific needs.

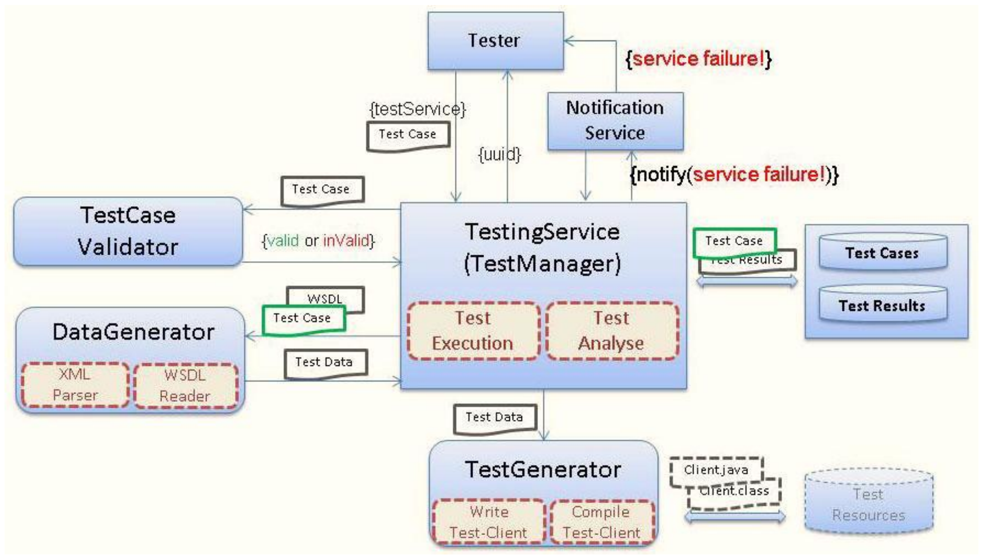


Figure 1. Architectural Layout of the Venice Testing Framework.

- *Different requirements:* different stakeholders have different perspectives on the platform, requiring support for a variety of use cases. On-demand testing and periodic testing should be supported in order to address the different needs of the service tester for the separate test cases.
- *Large amount of data*: the execution of a large number of test cases will produce a large quantity of data. The platform must be able to provide storage and analysis for a large quantity of test results.
- *Lack of trust:* access to testing data and results is a trust issues in the open environment of the Internet of Services. Stakeholders should be granted proper handling of the data they provide for testing purposes. Security mechanisms regulating the access to this data should be assured.
- *Lack of evolution control:* a service provider can change the functionalities of a running service at any time. This can result in an unexpected change for some of its users. In order to prevent this situation, service users affected by a change must be informed about service modifications.
- *Dynamicity*: the dynamic character of SOAs enables new services to be deployed on the platform, existing services to change, or removal of unused or defective services from the platform. The framework must automatically perform acceptance testing [8] on deployment of new services to ensure the quality of the resources offered on the platform. Regression tests [12] must be executed on every change of existing services to ensure compliance with existing SLAs and contract terms. Deactivation of test cases for deleted services should also happen automatically in order to prevent unnecessary resources usage.

V. SOLUTION ARCHITECTURE

The proposed testing framework enables Web service developers and other stakeholders to automatically and fully test services during development and run time. If an error occurs during the actual service execution (e.g., a service cannot be reached, or its output does not correspond to expected values) all participants of the testing process will be notified about this error.

For the usage of the testing framework two use-cases can be defined. A service developer can use the framework to check the functionalities of a service during development time. The framework also can be used to test the services at run-time. This use case scenario is useful especially for platform providers and service consumers. A monitoring service can navigate the testing framework to execute test cases on the basis of a predefined test schedule. Services can be tested on-demand or periodically.

The architecture of the framework is shown in Figure 1. The framework is composed of several components - *TestManager*, *TestCaseValidator*, *DataGenerator*, *TestGenerator*, a *database*, and a *repository* for test resources – which, in combination, execute the framework functionalities. It uses also the Notification Service of the Venice Framework [10] to keep all participants informed of test results.

A. Testing life-cycle

The framework supports all four phases of the defined testing life-cycle: test specification, test organization, test execution, test analysis. In the following, we describe functionalities of the framework components on the basis of these life-cycle phases.

1) Test specification

In order to execute a functional test, the testing framework needs some input data. This information should be defined by a service developer in XML. Our framework offers a XML schema to support the tester during the description of the test cases.

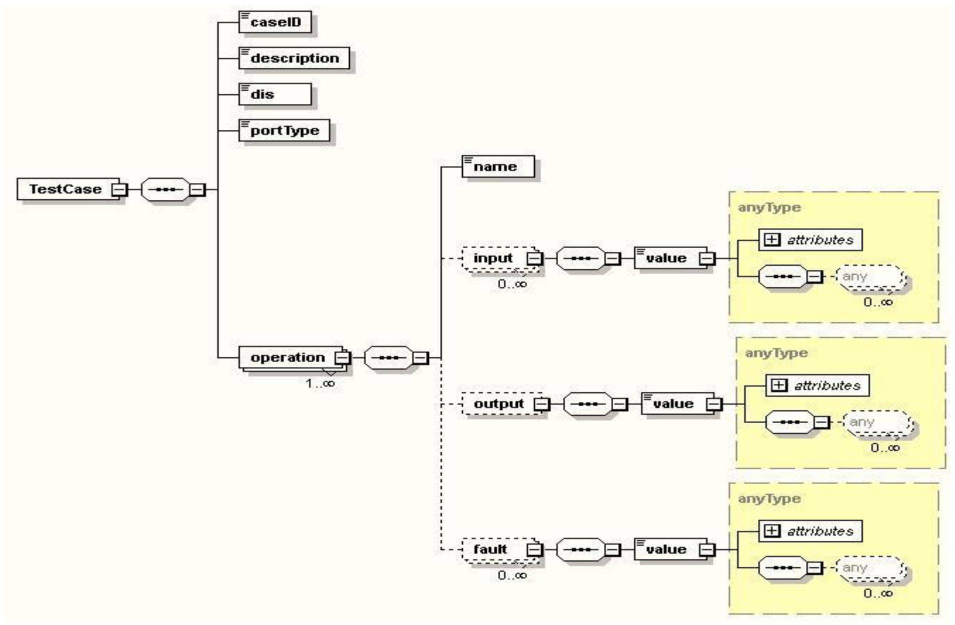


Figure 2. XML-Schema to define test cases.

Figure 2 presents the test case XML schema. A test case must have a unique name (*caseID*). The corresponding test case will be stored in the database with this name. Further important information in test cases are the Domain Information Service (dis) and port type (portType) fields. The dis provides meta-data for the service domain and enables service interaction in the Venice environment [10]. The portType defines an abstract name for a set of operations and messages. A test case has to define which operations will be tested along with its input and expected output. Each operation can also have a fault element, which should demonstrate a service call for an invalid input.

The framework offers operations to add test cases into a database, to get them and to delete them. Before storing into the database, the test cases have to be checked for their validity by the component *TestCaseValidator* (see Figure 1). Only valid test cases will be used.

2) Test organization

Different options for testing are offered to the tester. The tester can test all the functionalities of a service, meaning that all the port types implemented by the service will be tested. Platform providers can use this operation before the deployment of the new service on the platform for acceptance testing. Service consumer can test the entire functionality of a service through this operation.

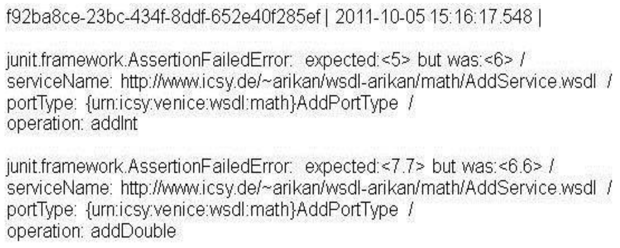
The tester can also test all the services which implement a certain port type. This operation is useful for a platform provider to perform automated tests for the complete platform. This also allows service consumers and platform provider to run performance tests or stress tests. Another useful operation is for creating a new test, which a service consumer can use to define a test case and then execute it.

3) Test execution

After a successfully validation of a test case against the test case schema (passing a syntax check), the test case will be parsed by the *DataGenerator* component of the framework, which also uses the WSDL description of the service to get more data. All test data (from the test case and the corresponding WSDL) will be encapsulated in a *TestCase* object and sent to the *TestGenerator*. The *TestGenerator* generates and compile a JUnit-based Java test. The resources are stored in a repository, which is created at the beginning of the testing process and deleted at the end of the testing process. After compilation, the newly generated test case will be executed.

4) Test analysis

If an exception is captured during the execution of the tests, this will be stored in a *TestResult* object. All test results will be written into the *TestResults* database (see Figure 3). These test results can be retrieved with the *getTestResult* operation of the Testing Service.



```
f92ba8ce-23bc-434f-8ddf-652e40f285ef | 2011-10-05 15:16:17.548 |

junit.framework.AssertionFailedError:  expected:<5> but was:<6> /
serviceName: http://www.icsy.de/~arikan/wsdl-arikan/math/AddService.wsdl  /
portType: {urn:icsy:venice:wsdl:math}AddPortType  /
operation: addInt

junit.framework.AssertionFailedError:  expected:<7.7> but was:<6.6> /
serviceName: http://www.icsy.de/~arikan/wsdl-arikan/math/AddService.wsdl  /
portType: {urn:icsy:venice:wsdl:math}AddPortType  /
operation: addDouble
```

Figure 3. An example to present the testing results for Add-Service.

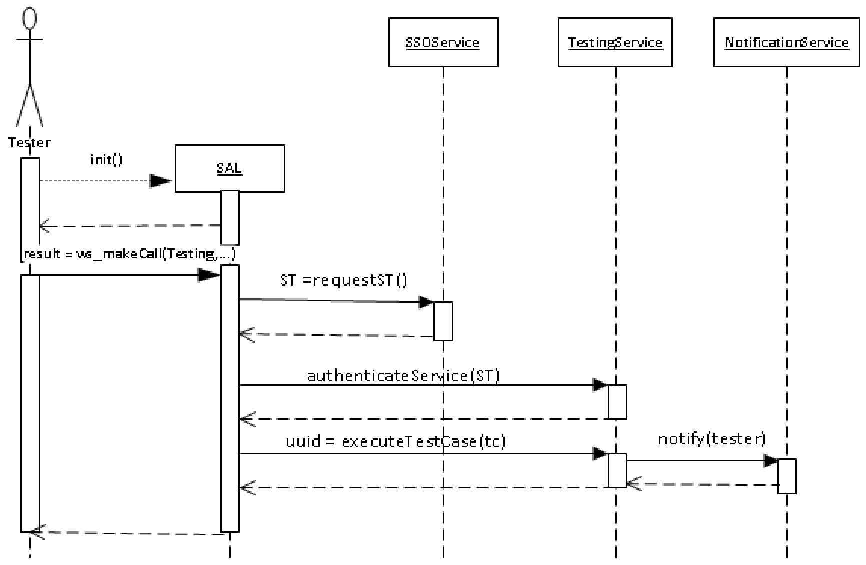


Figure 4. Use case diagram to demonstrate using of the Testing Service

B. *Implementation Prototype*

We implemented a prototype of the testing framework for the Venice (see Figure 4) platform. Venice is a SOA-based framework for building secure and dependable distributed applications; it supports service developers during developing, deployment, maintenance, and usage of Web services. Different service providers can use the Venice infrastructure to offer their services for service consumers. Figure 4 shows how the testing framework is used in the Venice environment. In order to use the testing framework, the tester first needs to initialize the *Service Abstraction Layer (SAL)* of Venice. The SAL accesses to additional functionalities like authentication and authorization, which are provided by the Venice Single Sign-On Service (SSO Service). To use the testing service, service consumers have to authenticate one time to the *SSOService*, which returns a service token (ST). The ST contains the authorization information that allows the user to prove his identity and to prove his right to access the testing service. All necessary operation invocations are made transparently for the service consumer. The next step is calling the desired operation of the testing service. After the testing process is finished, the tested service returns a unique *uuid*, which is used to request test results from the database. Service consumers will be informed of the test results through the notification service provided by Venice. Finally, test results are fetched from the database.

The testing framework is implemented in Java and uses the *JUnit* libraries. Figure 5 shows the implemented classes of the framework and their relationships.

The Testing class uses the *InputGenerator* to get input data as a *TestCase* object. The *InputGenerator* uses *DOMParser* to parse a test case, which was created by service developer in XML. The *DOMParser* class reads XML files and generates the corresponding input, output and fault objects, which will be added to a *TestCase* object. A *TestCase* object will be given back to the Testing class. It uses the *WriteUniTest* class to generate java test classes. These will be compiled, and then executed by *MyTestSuite*.

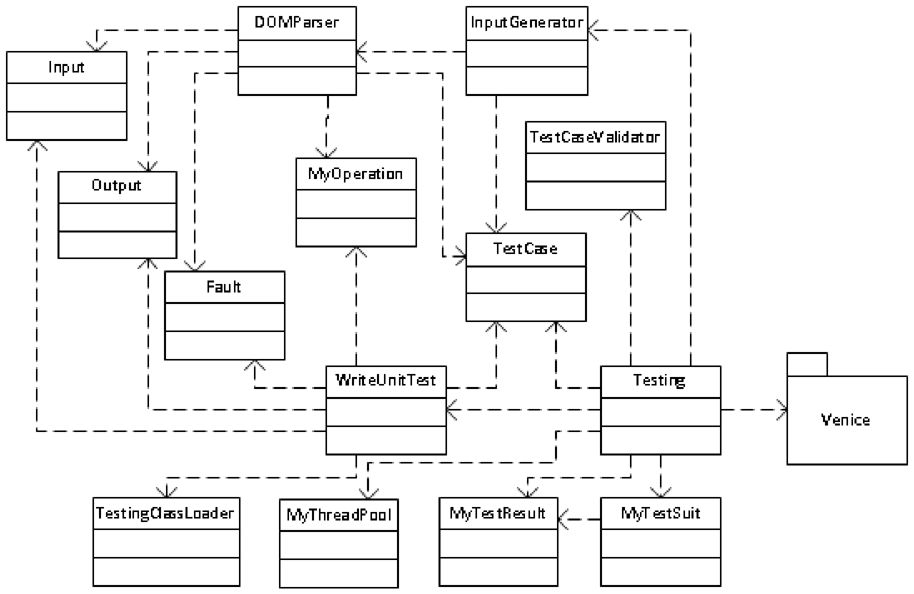


Figure 5. Static structure of the Testing Framework.

Test results will be added to a *MyTestResult* object and stored in the database. The clients will be notified through Venice's notification system. To perform incoming tasks more efficiently, we implemented a thread pool. Tests are temporarily stored in the *IncomingRequests* queue and are executed by worker threads in the thread pool.

VI. Conclusion and Future Work

In order to meet new quality requirements in software development, software testing has been researched for several years. With the application of SOA as a concept for development of distributed services on the internet –Internet of Services - new challenges for testing infrastructures were defined. In order to satisfy these challenges, we designed and implemented a generic testing framework. Our proposed framework is based on black-box testing and supports the whole testing life-cycle; from generation and checking of the test cases to compiling and execution of test cases.

This paper presented a generic testing solution and its prototype implementation for testing of IoS service platforms. In future, the functionality of the framework will be extended; we plan to enable the test result analysis and present statistics for executed test cases. Furthermore, in order to provide better performance and scalability, an asynchronous communication pattern will be implemented and integrated into the prototype. A graphical user interface is planned in order to increase usability.

VII. Acknowledgements

This work was funded by the German Ministry for Education and Research (BMBF project "iGreen" 01IA08005G). The authors are responsible for the content.

References

[1] N. Looker, M. Munro, and J. Xu, " Testing Web Services," the 16th IFIP International Conference On Testing of Communicating Systems, Oxford, 2004, unpublished.

[2] L. Frantzen, J. Tretmans, and R. Vries, "Towards Model-Based Testing of Web Services," Inter. Workshop on WS. - Modeling and Testing (WS-MaTe2006), Palermo, 2006, pp. 67-82.

[3] S. K. Chakrabarti, and P. Kumar, "Test-the-Rest: An Approach to Testing RESTful Web-Services," IEEE COMPUTATIONWORLD'09, Athens, 2009, pp. 302 – 308.

[4] E. Martin, S. Basu, and T. Xie, "Automated Robustness Testing of Web Services," Proceedings of the 4th International Workshop on SOA and Web Services Best Practices, Oct. 23, Portland, Oregon, USA., 2006, pp. 114-129.

[5] C. Bartolini, A. Bertolino, and E. Marchetti, "WS-TAXI: a WSDL-based testing tool for Web Services," icst, International Conference on Software Testing Verification and Validation, Colorado, 2009, pp. 326-335.

[6] soapUI, http://www.soapui.org, April 2012

[7] A. Bertolino, J. Gao, and E. Marchetti, "Automatic test data generation for XML Schema based partition testing, " IEEE Automation of Software Test, Minneapolis, 2007, pp. 4-11.

[8] H. Balzert, Lehrbuch der Software-Technik, Spektrum Akad. Verl., 1998, pp. 257.

[9] TEXO, Business Webs in the Internet of the Services, url: http://www.internet-of-services.com/index.php?id=276&L=0 , April 2012

[10] The Venice Service Grid, url: http://www.v-grid.info/html/pdf/The%20Venice%20Service%20Grid.pdf, April 2012

[11] T. Erl, Service-Oriented Architecture, Concept, Technology, and Design, Prentice Hall PTR, March 2009, pp. 290-291

[12] P. Liggesmeyer, Software Qualität – Testen, Analysieren und Verifizieren von Software, Spektrum Akad. Verl., Heidelberg, 2002.

[13] R. Pressman, Software Engineering: A Practitioner's Approach, McGraw Hill, Boston, 2001.

[14] P. Jalote, An Integrated Approach to Software Engineering, Springer Verl., 1997.

[15] S. Arikan, M. Hillenbrand, and P. Müller, "A Runtime Testing Framework for Web Services", 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'10), Lille, France, 2010.

[16] IEEE Standard Glossary of Software Engineering Terminology, IEEE std 610.12-1990, September 1990, url: http://web.ecs.baylor.edu/faculty/grabow/Fall2011/csi3374/secure/Standards/IEEE610.12.pdf, April 2012

[17] C. Haubelt, J. Teich, Digitale Harware/Software-Systeme - Spezifikation und Verifikation, Springer Verl., 2010, pp.95-111.

[18] A. Kabzeva, M. Hillenbrand, P. Müller, and R. Steinmetz, "Towards an Architecture for the Internet of Services", 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'09), Patras, Greece, 2009.

[19] C. Janiesch, R. Ruggaber, and Y. Sure, "Eine Infrastruktur für das Internet der Dienste", HMD - Praxis der Wirtschaftsinformatik, 45(261):71–79, June 2008.

[20] J. Spillner, M. Winkler, S. Reichert, J. Cardoso, and A. Schill., "Distributed Contracting and Monitoring in the Internet of Services", Ninth IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2009), vol. 5523 of Lecture Notes in Computer Science, pp. 129–142. Springer-Verlag, Berlin Heidelberg,2009.

[21] B. Sosinsky, Cloud Computing Bible, Wiley Publishing, Inc., 2011.

[22] OASIS, SOA-EERP Business Service Level Agreement Version 1.0, Commetee Draft 03, January 2010, url: http://docs.oasis-open.org/soa-eerp/sla/v1.0/SOA-EERP-BSLA-spec-cd03.pdf, April 2012