ARTIST

FP7-317859

Advanced software-based seRvice provisioning and migraTIon of legacy SofTware

Deliverable D11.2

Methodology and architecture for end user-based testing M24

Editor(s):	Patrick Neubauer, TUWIEN Javier Troya, TUWIEN
Responsible Partner:	Technische Universität Wien
Status-Version:	v1.2
Date:	2014-09-25
Distribution level (CO, PU):	PU

BY-SA

Project Number:	FP7-317859
Project Title:	ARTIST

Title of Deliverable:	D11.2 Methodology and architecture for end user-based testing M24
Due Date of Delivery to the EC:	30/09/2014

Workpackage responsible for the Deliverable:	WP11 - Migrated product testing, validation and certification
Editor(s):	Patrick Neubauer, TUWIEN Javier Troya, TUWIEN
Contributor(s):	–
Reviewer(s):	Javier Canovas, INRIA
Approved by:	All Partners
Recommended / mandatory readers:	WP8, WP9, WP11

Abstract:	In this deliverable, a methodology and an abstract as well as concrete architecture for the ARTIST use cases is developed which allows to run the original application and the modernized application in parallel to reason about behavioural equivalence.
Keyword List:	end-user testing, user session-based testing, web service testing
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/

Document Description

Document Revision History

Version	*Date*	*Modifications Introduced*	
		Modification Reason	*Modified by*
v0.1	01/06/2014	Table of Contents	TUWIEN
v0.2	14/06/2014	Related Work Section completed	TUWIEN
v0.3	14/07/2014	Implementation Section completed	TUWIEN
v0.4	31/07/2014	Delivery and Usage Section completed	TUWIEN
v0.5	26/08/2014	Initial version of document	TUWIEN
v1.0	03/09/2014	Improvement of final version	TUWIEN
v1.1	17/09/2014	Incorporation of reviewer suggestions	TUWIEN
v1.2	25/09/2014	Added section dedicated to innovations	TUWIEN

BY-SA

Table of Contents

Table of Contents 4
Table of Figures 5
Table of Tables 5
Terms and Abbreviations 6
Executive Summary 7
1 Introduction 8
1.1 About this deliverable 8
1.2 Innovation and novelties 9
1.3 Document structure 10
2 Related Work 11
2.1 Web application testing 11
2.2 User session-based testing 13
2.3 Relations to our approach 15
3 Implementation 16
3.1 Fitting into overall ARTIST solution 16
3.2 Functional description and methodology 18
3.2.1 User Request Discovery 19
3.2.2 User Request Mapping and Execution 20
3.2.3 Runtime Trace Extraction 21
3.2.4 Runtime Trace Abstraction 21
3.2.5 Runtime Trace Comparison 22
3.3 Technical description 22
3.3.1 Prototype architecture 23
3.3.2 Testing Monitor 23
3.3.3 User Request Discovery 24
3.3.4 User Request Mapper 25
3.3.5 Virtual User 25
3.3.6 Trace Extractor 26
3.3.7 Trace Abstractor 26
3.3.8 Test Oracle 27
3.4 Technical specification 27
4 Delivery and Usage 28
4.1 Package Information 28
4.1.1 Prototype Module 28
4.1.2 Migration Trace Model Module 28
4.1.3 Exemplary Migrated Web Service Module 28
4.1.4 Exemplary original Web Service Module 29

BY-SA

4.2 Installation Instructions 29
4.3 User Manual 29
4.4 Run Test Cases 29
4.5 Build Test Cases 32
4.6 Licensing Information 37
4.7 Download 37
5 Conclusion 38
6 References 40

Table of Figures

FIGURE 1: Migration Process of ARTIST 8
FIGURE 2: Software Migration into the Cloud 17
FIGURE 3: Methodology for end user-based testing 19
FIGURE 4: Web service migration trace meta model 21
FIGURE 5: Architecture of end user-based testing 23
FIGURE 6: Running the Maven install inside Eclipse IDE. 31
FIGURE 7: Running the JUnit tests inside Eclipse IDE. 32
FIGURE 8: Exemplary SOAP request envelope referenced in step 3. 33

Table of Tables

BY-SA

Terms and Abbreviations

APFD	Average Percent of Faults Detected
CCFG	Composite CFG
CFG	Control Flow Graph
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
FSM	Finite State Machine
ICFG	Inter-procedural Control Flow Graph
IDE	Integrated Development Environment
MC/DC	Modified Condition Decision Coverage
NFR	Non Functional Requirements
OCFG	Object CFG
SbSp	Service based Software Providers
SOAP	Simple Object Access Protocol
WATM	Web Application Test Model
WSDL	Web Service Description Language
XML	Extensible Markup Language

CC BY-SA

Executive Summary

This document represents Deliverable D11.2, i.e., the methodology and architecture for end user-based testing, and hence represents the documentation of the ARTIST end user-based prototype. The basic methodology starts by recording a user request that is sent to the modernized system and to map it into one or multiple corresponding original requests. Both the migrated user request and the original user request are then sent to their respective application for immediate execution. In the next step the execution trace is extracted from their respective application and abstracted into a platform independent or platform-specific compatible level. Once arrived at this high level of abstraction, a Test Oracle compares both execution traces and yields a verdict that describes if the behavior of the original user request(s) differ from the migrated user request. The prototypical solution that has been built in the extent of D11.2 is based on web service testing. In particular, the SOAP web service technology has been used as an exemplary web service technology. While SOAP web services are used by at least two use case providers, they are also platform independent. Hence, even if use case providers apply different programming languages, the exemplary prototype, that builds on the SOAP technology, can be used in any case as it is executed externally from the use case application. Overall, the end user based testing methodology, earlier discussed in WD11.2, is feasible in case of web services and in particular in case of the SOAP technology. Therefore, the outcome of this work represents an interesting case to further look into from the perspective of other web service technologies, such as REST, or even non-web service related applications.

BY-SA

1 Introduction

In the context of the ARTIST project, the migration is performed based on model-driven engineering techniques [3]. Thus, the original software is reverse-engineered to obtain a model-based representation in terms of platform-specific models. These models are transformed into more abstract models, such as platform-specific compatible models or platform-independent models (e.g., UML models), which describe the original application in a platform-independent or platform-specific compatible way. The actual migration is performed by applying model transformations and code generators to create the migrated software. After the migration has been accomplished, we have to verify the behavioral equivalence of the original software and the migrated software.

In the post-migration phase of a software migration project (i.e., when original software is migrated to new technologies or platforms), one of the tasks to ensure software quality is to evaluate if the migrated software still conforms to the original specification. Hence, the migrated software, by taking into account any expected improvements as, e.g., in terms of performance, is assessed to be still meeting the initially defined behavior.

1.1 About this deliverable

In this document we focus on assessing the behavioral aspect of the migrated software with the original specification. Therefore, the investigation is based on the behavioral equivalence of the original software and the migrated software. In other words, it is evaluated whether both the original software and the migrated software behave similarly in terms of their functional requirements. In more detail, this document focuses on the end user-based approach to verify behavioral equivalence by introducing the prototype built within the means of Task 11.2, Methodology and architecture for end-user based testing. The context of this task is shown in Figure 1. The test case based approach for testing behavioral equivalence, i.e., Task 11.1, was already described in the working document WD11.1.

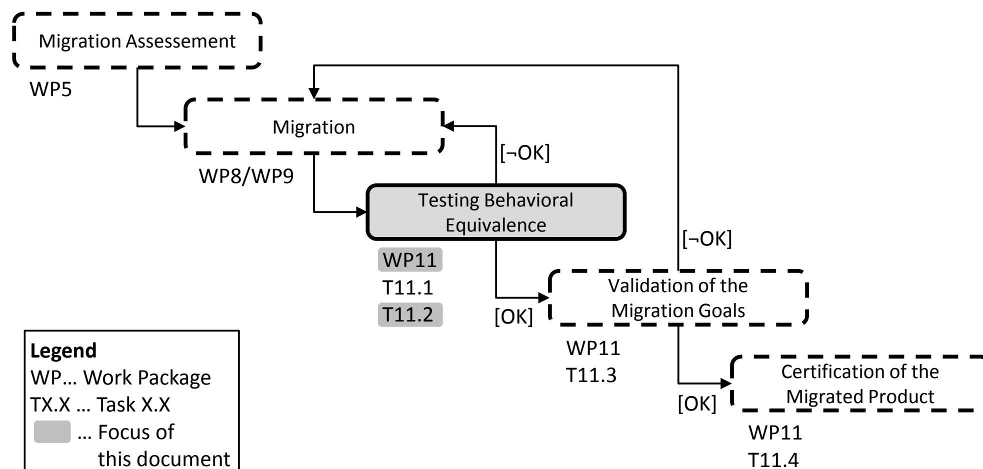


Figure 1: Migration Process of ARTIST

BY-SA

The goal of this document is to describe the basic methodology and architecture used for end used-based testing to test behavioral equivalence. We define end user-based testing in terms of web services to which real web service requests are issued to test the migrated application. Both the original system web service as well as the migrated system web service run in parallel and the interaction of the user with the migrated system are recorded and ad hoc equivalent interaction is executed on the original system. A controlling component monitors both systems and verifies whether there is a deviation in the behavior or both systems behave equally. Both systems can run in parallel either locally and/or in the cloud and their behavioral monitoring is performed in an automated manner after the monitoring setup has been established. The overall idea is not to provide a tool to cover all kinds of different end user-based testing scenarios but to provide an architecture that could potentially be used in a large set of scenarios and to focus on a specific scenario for evaluation purposes. The scenario chosen evaluates the behavioral equivalence between a migrated SOAP web service and its corresponding original SOAP web service.

1.2 Innovation and novelties

As introduced above, this document is dedicated to the presentation of the implementation of the end user-based testing components as used and/or developed in the context of the ARTIST project and use cases.

At the universal level, the main innovative aspect of this ARTIST end user-based testing prototype is its objective of providing an evaluation on the feasibility of the proposed architecture. The architecture combines ideas and insights from several approaches [5, 12, 15] into a novel gray-box testing approach. A discussion of the relationship with other approaches can be found in section 2.3.

To be more specific, within the architecture of our solution, both the migrated application and the original application run in parallel and retrieve user requests in parallel. Together with this feasibility and the automatic abstraction made on the resulting application behavior produced by the individual applications, our approach allows to compare intentionally equal user requests on two different applications. While the original user request is automatically built from the migrated user request, the original response and the migrated response are abstracted into a comparable format. These automatic transformations and abstractions, which are performed by the prototype, can be created thanks to the information gained from the migration trace model. The latter model typically emerges by migrating an original application to a migrated application and hence represents the changes performed on the original application that result in the migrated application.

To summarize, the main innovation is represented by the architecture that has been developed and includes the execution of a novel gray-box testing technique that allows to perform parallel behavior equivalence testing on two distinct applications by taking into account their migration trace. Furthermore, while the prototypical implementation proves the feasibility of the approach, it also shows that the architecture could potentially be used

BY-SA

in a large set of scenarios. This means that not only SOAP web service applications, but other kinds of applications (e.g., REST web services or even desktop GUI applications), could be tested on behavioral equivalency by developing a solution based on the proposed architecture.

1.3 Document structure

The document is structured as follows. Section 2 presents related work in the area of web application testing and user session-based testing. Section 3 depicts the prototypical implementation in terms of a functional and technical description. The technical description itself is composed of the prototype architecture and its components. Section 4 describes the delivery and usage of the prototype in terms of package information, installation instructions, user manual and so forth. In Section 5 the document is shortly summarized, concluded, and future work is proposed.

BY-SA

2 Related Work

In this section we explore two areas of related work that are also concerned with testing. First, there is web application testing, which shares a lot of features with cloud applications, like the way they communicate with the user. For example, applications running in the Google App Engine can communicate with other applications or access other resources by using HTTP (port 80) or HTTPS (port 443) requests and responses. Allowed methods include `GET`, `POST`, `PUT`, `HEAD`, and `DELETE`[1], similar to most web applications. As a second area there is user session-based testing, which is a technique that is often deployed in the area of web application testing. In user session-based a session consists of all requests of a specific user within a certain time frame and session data is the basis from which test cases are generated.

2.1 Web application testing

Di Lucca and Fasolino [3] give a good overview of the different aspects of web application testing. Their work is summarized in this paragraph. A web application is defined as a distributed system, with a client-server or multi-tier architecture that runs in a heterogeneous execution environment, e.g., hardware, operating systems or browsers, can have multiple distributed users that use the application concurrently, allows the use of components developed in different technologies, e.g., programming languages, and can create new components at run time depending on the user inputs and the server status. In our opinion, each of the aspects described can also be true for cloud applications. Di Lucca and Fasolino [3] further distinguish between testing non-functional requirements (NFR) in the area of web applications. NFR are mainly affected by the running environment of the application, e.g., server or network infrastructure, whereas functional requirements are mostly realized in the application itself, e.g., the source code. Different types of NFR testing include performance testing, load testing, stress testing, compatibility testing, usability testing, accessibility testing and security testing. Functional requirements specify the functionality of the web application and rely on a test model representing the relationships between components. Test models can be separated in *behavior models* describing the functionality irrespective of its implementation, e.g., use case models, decision tables or Finite State Machines (FSM), and *structural models* describing the data and control flow of the implementation, e.g., control flow graphs (CFG). Testing of functional requirements can be performed on different levels of detail, such as unit testing, verifying the behavior of client pages or server pages, integration testing, checking how the pages work together, and system testing calculating different coverage metrics, e.g., use case coverage or link coverage. We can further categorize three different test strategies where each relies on different information. Black-box testing strategies consider the functionality stated in a requirements specification, white-box testing strategies use knowledge about the inner workings of the implementation, i.e., the source code, to verify coverage criteria, and gray-box testing strategies are a mixture between white-box testing and black-box testing and

[1] https://developers.google.com/appengine/docs/java/urlfetch/overview

BY-SA

are used to test against a given specification by using some knowledge about the internal workings.

In the following paragraphs, a prominent example of each testing strategy is explained shortly. For details on the approaches, please see the referenced literature.

Liu et al. [11] propose a white-box testing strategy using the Web Application Test Model (WATM) they developed. The WATM consists of two sub models, the object model and the structure model. In the object model each entity of a web application is created as an object and relationships between those objects can be defined. There are three possible types of objects, client page, server page, and component, and seven types of relationships, i.e., inheritance, aggregation, association, request, response, navigation and redirect. The object model is represented as an Object Relation Diagram. The structure model holds the data flow information of the web application and is represented in four different graphs, each responsible for a different kind of data flow level. The control flow graph (CFG) shows the data flow information of an individual function, the inter-procedural control flow graph (ICFG) depicts the data flow involving more than one function, the object control flow graph (OCFG) is used to connect all ICFGs and CFGs within an object to obtain the def-use chains, and the composite control flow graph (CCFG) describes the data flow information between client and server pages. After extracting the data-flow information from the WATM, test cases are derived on five different levels: function, function cluster, object, object cluster, and application. Def-use chains are used to generate test data that can satisfy a given test criteria and to uncover definition/use anomalies. A def-use chain is a data structure that holds information about a variable and its read- and write-access. The effectiveness of the approach is shown in a small example.

Andrews et al. [1] propose a black-box testing technique on system level using logical web pages, which are clustered into FSMs. FSM hierarchies are used to partition the models into more manageable parts, whereas the topmost FSM represents the whole application and the lowest FSMs represent single pages or parts of pages. Each FSM therefore models a subsystem of the web application and generates subsequences of the web application states. These states can be seen as test requirements. By combining those subsequences, complete executable test cases are created, which are then executed against the web application under test. To avoid state space explosion additional constraints on the order and requirements of inputs are used to reduce the input value set. A prototype is implemented to support the technique.

Elbaum et al. [5] propose using user session data to test web applications as a gray-box testing approach. Since this technique uses data similar to the data we gather from the interaction of the user with a cloud application, or more specifically a user with a migrated web service running in the cloud, we will present user session-based testing in more detail in its own section.

BY-SA

2.2 User session-based testing

In user session-based testing logged interaction of the user with the (web) application is used for test case generation. A simple form of user session-based testing can use a capture/replay tool [7] to record the interactions of a test user with the application and replay the collected session in an automated manner without the user. Using session data for testing makes use of the fact that the ability of recording user requests is often built into web servers [15].

Alternative strategies to collect user interaction include among others the approach by Orso and Kennedy [12], who implemented a custom class loaders for Java to instrument the necessary code for logging, or the approach of Steven et al. [17], who implemented the jRapture tool that uses a modified version of the Java API classes to capture and replay program executions.

One of the first studies about user-session based testing was done by Elbaum et al. [4, 5], who shows that user session testing can be as effective in terms of fault detection as existing white-box testing techniques. It is noted however that user session-based testing should be used complementary to existing techniques as it usually detects different classes of faults. Such classes include among others scripting faults, web page faults and database query faults. A user session is defined as a collection of client requests of one user, whereas each client request is represented as an URL, also called base request by [15], and a number of name-value pairs. One example for a user request URL is `view_courses.jsp?name=Testing&semester=2013S`, where the base request is `view_courses.jsp` and two name-value pairs, i.e., *name* with the value of `Testing` and *semester* with the value of *2013S* are present. To generate test cases from user sessions, Elbaum et al. present four approaches: sequential replay of each session separately, replaying a mixture of interactions, replaying multiple sessions in parallel to check concurrency, and mixing different requests with additional known problematic requests created by hand. In the example approach shown by Elbaum et al. all user requests were collected and user sessions were randomly selected for creating a test suite. A problem that can arise when using user session data for testing might be that the outcome of a request might not only depend on the data given by the URL, but also on the internal state of the application. The example given by Elbaum et al. [5] is a reservation request, that might function differently depending on the number of available seats. To resolve this issue you can either associate snapshots of the internal state with requests or ignore the state and use the session data to provide an effective partitioning heuristics of the possible inputs. Privacy issues can be reduced by capturing the user requests on the server side and not running scripts on the client side. Another problem stated by [9] can occur when trying to consider all name-value pairs of a request, because the number of pairs can vary heavily depending on the request and not all algorithms are equipped to deal with such variation.

As stated before, the approach of Elbaum et al. [5] uses randomly selected user session data and does not consider the size of the test suite, which might be infeasible for real world

BY-SA

applications. One way to reduce the test suite size was proposed by Sampath et al. [15]. Their approach uses concept analysis, where objects, i.e., user requests, are clustered based on discrete attributes. To keep the number of attributes low, only the URL and the corresponding session id are used as attributes, without the associated name-value pairs. Using the incremental concept formation algorithm developed by Godin et al. [6], a set of initial user session data can get updated stepwise by analyzing additional user sessions. To apply concept analysis, a prototype framework was developed that executed the following summarized phases automatically: web application execution/logging, test case generation and reduction, test coverage evaluation, replay of the reduced test suite, generation of a coverage report, and incremental update of the test suite. As a result, it was found that the statement and function coverage of a reduced test suite is almost as good as the coverage of the corresponding full test suite. Due to the exclusion of the name-value pairs, however, about a fifth lesser faults were detected.

Further developing this approach and trying to improve the fault detection rate again, Sampath et al. [14] introduce a prioritization of the generated test cases. Prioritizing test cases means to schedule the test cases according to some criterion to satisfy a performance goal. Examples for such criteria include among others modified condition decision coverage (MC/DC), requirements coverage, cost estimates, and event coverage. In the context of user session-based testing, Sampath et al. examined the following criteria: test length based on number of base requests (ascending and descending), most frequent accessed pages, unique coverage of parameter-values, 2-way parameter-value interaction coverage, test length based on number of parameter-values (ascending and descending), and random permutations. Three applications with different characteristics that have previously been discussed by Sampath et al. [15] and Sprenkle et al. [16] are evaluated with respect to their fault detection rate, their average percent of faults detected (APFD) proposed by Rothermel et al. [13], and their test suite execution time. As a result, no prioritization criteria is clearly better than the others in all cases, although some techniques that consider parameter-value counts or interactions seem to be faster in detecting the faults. Particularly 2-way prioritization works very well for two out of the three applications and has the highest APFD overall.

Li et al. [9, 10] used the k-Medoids algorithm to cluster user sessions and reduce the original test suite size. In contrast to Sampath et al. [15] not only the base request is used to reduce the test suite, but also the name-value pairs are considered as well as the ordering of the requests within a session. The necessary non-numeric user session data is extracted from the server logs in the form of name-value pairs. To achieve a test suite reduction, the k-Medoids clustering algorithm is applied to cluster user sessions into disjoint sets, or clusters. In a second step user sessions are picked randomly from each cluster as representatives. The algorithm is configured in a way that user requests in the sessions within one cluster have a similar number of name-value pairs. To measure the suitability of that kind of clustering for web applications, the inner distance of one cluster (IDC) and the outer distance of two clusters (ODC) are calculated as well as function coverage and statement coverage are used as metrics. The clustering algorithm is considered success-

BY-SA

ful when the IDC is smaller than the ODC and the approach is suitable when the coverage values of the reduced test suite are not considerably smaller than of the original test suite. In an empirical study Li et al. showed the effectiveness of their approach and the algorithm for web applications.

2.3 Relations to our approach

In our prototype, we implemented the “replay” part of Sampath et al.’s approach [15] since our test case reads and further elaborates on a SOAP envelope from a file essentially representing a captured user request. However, it represents a single interaction (not multiple) as it may be mapped to multiple original user requests (e.g., the migrated operation “createUser” represents a consolidation of the original operations “createUser” and “updateUserAge”) from which both produced results are then evaluated by the test oracle.

When looking into Orso and Kennedy’s approach [12], we did consider the use of Aspect Oriented Programming, but due to the fact that it requires modifying the use case application by adding code we decided against this approach, such that our tool stays independent from the use case application. Hence, it is able to deal with web services running on any kind of web server based on .NET, Java or such technologies.

The methodology in Task 11.2 has been built on the premises of Elbaum et al.’s outcome [5] that user session based testing, a gray box testing technique, can be as effective in terms of fault detection as existing white-box testing techniques and on the fact that the overall ARTIST methodology already incorporates white box testing in Task 11.1. Hence, Task 11.1 and 11.2 represent a complementary testing technique to detect different classes of faults. To be more specific about Task 11.2 and a comparison with Elbaum et al., in our approach a user session is composed out of elements (i.e., web service operations), element parameters (i.e., web service operation attributes), and parameters values (i.e., web service operation attribute values). The issue that arises in Elbaum et al.’s approach related to the fact that a user request depends on the internal state of an application exists also in our approach. We resolve this issue by starting out our test cases from equivalent application states. Hence, both the original application and the migrated application have to be in an equivalent state at the point in time when test cases are issued by our prototype. In order to compare our approach with the results found by Godin et al. [6], or results from other web service based testing approaches, further investigation is required.

Using the knowledge gained by examining the two areas, web application testing and user session-based testing, we defined a methodology to realize end user-based testing in the context of ARTIST. This methodology is presented in the next section alongside the overall description of the prototypical implementation.

BY-SA

3 Implementation

3.1 Fitting into overall ARTIST solution

The goal of applying end user-based testing in ARTIST is to test the behavioral equivalence of the migrated software with respect to the original software by having the user generate real world test cases. In the following we outline briefly the migration process of ARTIST and put the functional behavior equivalence into the context of the overall migration process and then discuss the notion of behavioral equivalence.

Migration scenario in ARTIST. When migrating original software into the cloud, we can distinguish between different migration cases from a conceptual point of view. These migration cases are depicted in Figure 2. In Figure 2a, software is simply deployed into the cloud, whereas only the original runtime infrastructure is replaced by a cloud-based runtime infrastructure and the software itself remains untouched. Thus, only the deployment descriptor is affected. In such a scenario, it is not necessary to test the behavioral equivalence of the original software and the migrated software because the behavior has not been changed. It has to be verified, though, whether or not the deployment has been successful. In the second scenario, depicted in Figure 2b, not only the runtime infrastructure is changed, but also the software itself is affected, for instance, to further exploit specific cloud-based platform features. As the software itself is affected, the behavioral equivalence has to be verified. In this case, however, the only specification of the intended behavior of the migrated software is encoded in the source code of the original software itself, not considering existing test cases or specifications; thus, the implementation is at the same time the only specification of the expected behavior. The lack of a more explicit and abstract specification, as well as the fact that a behavior specification using general purpose programming languages may exhibit (possibly infinitely) many more states than strictly needed to specify the intended behavior, hampers the analysis and verification of the expected behavior in the migrated software significantly.

In ARTIST, we defeat this issue by deriving model-based specifications of the expected behavior, which is the third scenario depicted in Figure 2c. More precisely, the migration is performed in two or more abstraction steps and subsequent specialization steps: from the original software, first a platform-specific model (original PSM) is derived, which raises the level of abstraction from source code to a more abstract modeling language (such as UML). However, this PSM still contains information about the specifics of the original platform. Therefore, another abstraction step is applied to obtain a platform independent model (PIM), which describes the structural and behavioral aspects of the software without making any assumptions on the respective platform, be it the original runtime infrastructure (RTI) or the targeted cloud-based RTI. Once this PIM is available, the actual migration is applied by specializing the specification of the software towards the cloud-based platform resulting in a migrated PSM, which finally may again be refined to the migrated software. Additionally, a new deployment descriptor has to be generated.

Abstractions and Specializations. As mentioned before, the PSMs are abstractions of

BY-SA

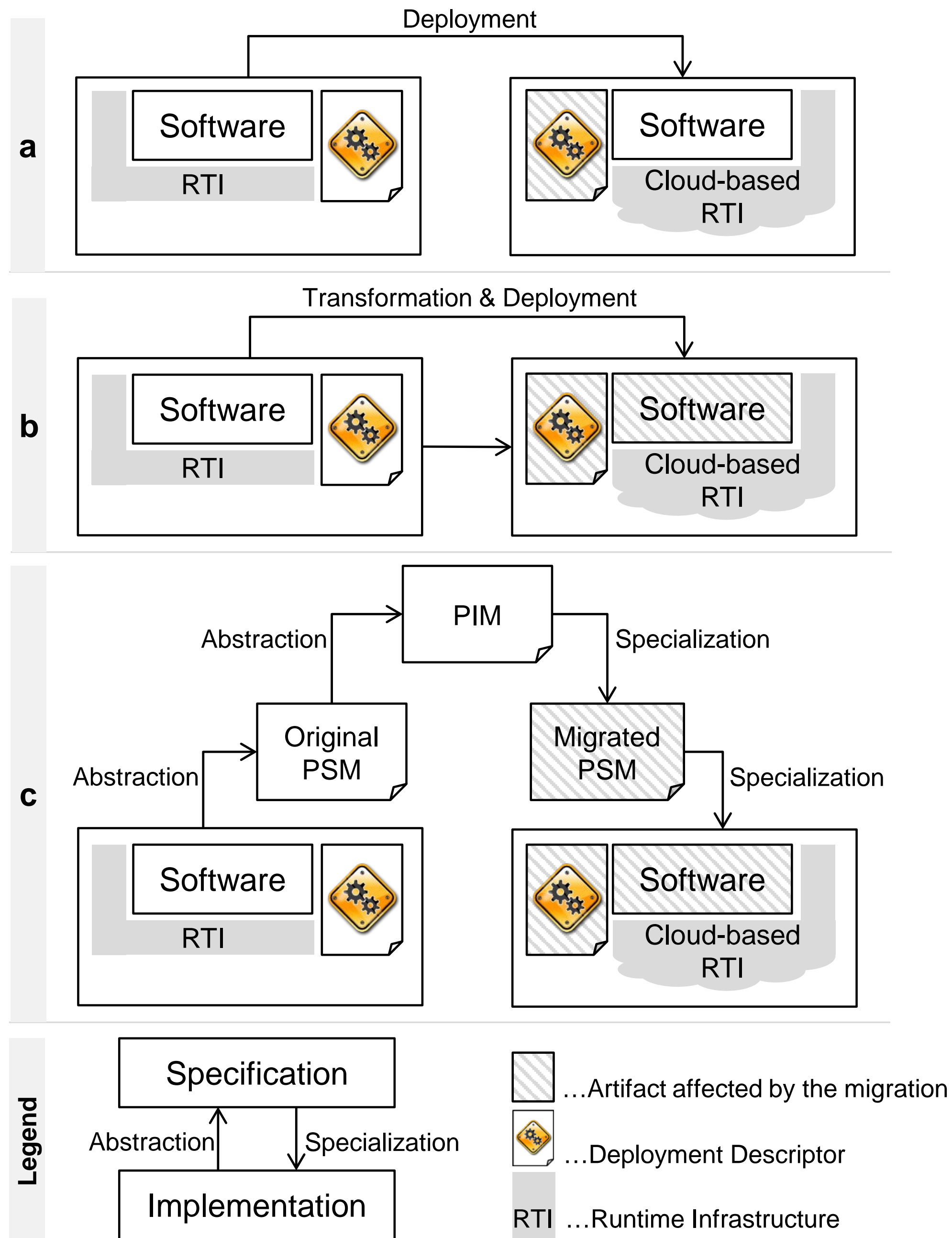


Figure 2: Software Migration into the Cloud

the respective software and the PIM is again an abstraction of both PSMs (cf. Figure 2c). In reverse, the respective software is a specialization of the respective PSMs, which are again specializations of the PIM. By definition, a specialization must realize all properties, both structural and behavioral, that are defined in the respective abstract artifact. This means, abstract artifacts serve as specifications for the specialized artifacts and the specialized artifacts are implementations of their specification (cf. legend of Figure 2). This property of the relationship between abstractions and specializations can be exploited for testing the behavioral equivalence of the migrated software with respect to the original software without having to compare the behavior of the source code artifacts with each other directly as discussed in the following.

BY-SA

Behavioral Equivalence. Many different notions of behavioral equivalence have been introduced in the various areas of computer science, such as program and software verification, database theory, and software design. In general, two things are equivalent if they are "regarded as mutually compensating each other, or as exchangeable" [8]. The key notion of equivalence here is *exchangeability*. In other words, equivalent things are indistinguishable to an observer and can be exchanged without affecting the observable behavior. However, the notion of equivalence differs regarding the notion of observer in different areas of computer science (i.e., what is the observer and which level of detail can the observer monitor). In particular, this general notion has been refined to more specific meanings. For instance, in program verification, a program can be trace equivalent, input-output equivalent, weakly and strongly bisimilar, etc. In database theory, other classes of equivalence have been defined for relations, such as behavioral equivalence, mapping equivalence, and transformational equivalence [2]. Thus, it is difficult to find a unified definition of the notion of equivalence for all possible use cases of ARTIST. Moreover, the equivalence problem of complex programs is considered to be undecidable and unsolvable in many cases [18]. As ARTIST aims to deal with large real-world software systems, it seems to be unfeasible to verify full equivalence formally. Therefore, we relax the notion of behavioral equivalence in the context of ARTIST for the sake of practicability and use this term to refer to *ad hoc equivalence*, which denotes an equivalence relation that is "appointed to or for some particular purpose" [8]. This definition will always satisfy the core assumption of equivalence (i.e., exchangeability), but the particular meaning is adapted and aligned to be most appropriate in a practical sense for the specific migration goals and strategies, as well as for the source and target technologies.

3.2 Functional description and methodology

After clarifying how end user-based testing fits in the context of the overall ARTIST solution in Section 3.1, in this section we present the functional description and methodology of the end user-based testing prototype that evaluates the behavioral equivalence of a migrated software in comparison to an original software.

This methodology is based upon the principles of MDE and abstraction. The data created by the user when interacting with the migrated software (i.e., the user issuing requests to the web service) is used as base to generate and validate a test case. Briefly, each request the user performs on the migrated software is recorded and an ad hoc equivalent request is performed on the original software. In order to verify whether both software applications have similar behavior, the output (i.e., the output produced by issuing a web service request) produced by both applications are compared. To allow such a comparison it is necessary to bring the execution traces (i.e., the web service response) containing the behavior of both applications into the same level of abstraction. Therefore, trace models (i.e., models that capture the migration from the original system into the migrated system in form of a trace) are used to abstract the execution traces into a platform-independent representation. This approach alone might not be sufficient to prove behavioral equivalence since the detailed behavior of the methods, i.e., the method body, could be different.

BY-SA

However, in combination with the test case-based approach examining that method behavior, this methodology can be determined adequate. In terms of classification, we can assign the methodology to the gray-box testing techniques as it uses both the migration trace, i.e., the trace model produced during the migration process, as well as knowledge about the output produced, i.e., the execution traces in form of web services responses, as basis.

The methodology is divided into five consecutive steps as depicted in Figure 3. The first two steps are used to bring both, the migrated software and the original software, into the same state by executing the same request. The last three steps are concerned with retrieving the execution traces, the abstraction of the traces, and the comparison of the traces. Each step is described in the following sections. Please note that the methodology is concerned with automatic verification of the behavioral equivalence between the original system and the migrated system and that therefore new features (e.g., operations that cannot be traced back to the original application) not covered in the original system must be checked manually or by providing additional information.

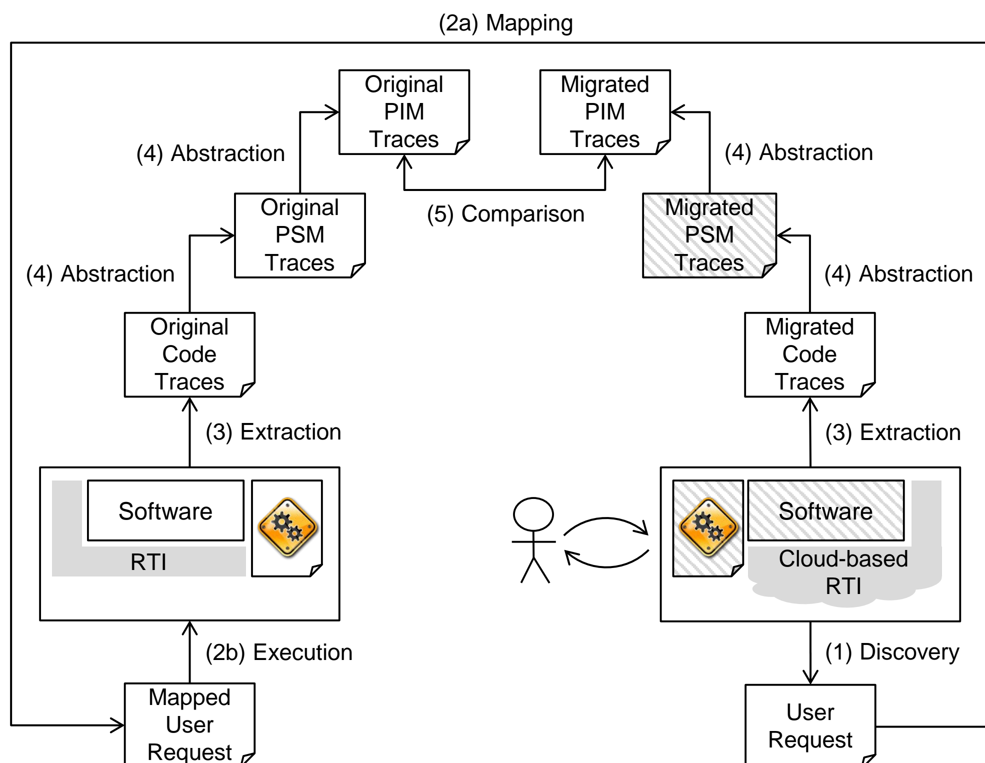


Figure 3: Methodology for end user-based testing

3.2.1 User Request Discovery

The initial step in the ARTIST end user-based testing methodology is the User Request Discovery in which the user interacts with the migrated software in form of user requests. Each of these user requests corresponds to one action at the migrated software (i.e., on the cloud side), for example by retrieving data from the database or manipulating objects

BY-SA

used by the application. In more detail, web service requests cause the application, according to the first example mentioned before, to retrieve data from the database and build a corresponding web service response that incorporates the result retrieved from the database. A user request could be, e.g., a HTTP request in terms of a POST or GET request in a web application, or a UI-specific request on top of a GUI desktop application.

For web application based systems we envision two approaches, the online approach and the offline approach. On one hand, in the online approach, user requests are recorded directly by using a proxy between the user and the cloud. Each user request gets processed (i.e., mapped) and then forwarded to the target cloud application. Here, the processing time of the request should be kept as short as possible to reduce the waiting time for the user. On the other hand, the offline approach is to use the logging mechanism of the cloud application or load balancer or introduce such a mechanism if not currently present. In this case, all user requests are recorded in a log file and traversed at a later point in time. Feasible methods to generate this log file might include the introduction of additional code into the respective methods, the use of aspect-oriented programming, or, at a higher level, the utilization of the load balancers log mechanism.

In the prototypical solution we directly retrieve user requests from the user, process (i.e., map) the request, and then send it to the desired applications. Hence, this solution represents the “online approach” as it incorporates a proxy service retrieving both information about the request as well as the request itself and forwards the request to the target applications. The latter proxy service is incorporated in the Testing Monitor described in Section 3.3.2. While the retrieved request itself can be as-is forwarded to the migrated application, a original request has to be built within the prototype that fits the original application before it can be forwarded to the latter application. The latter request is built during the “user request mapping” step.

3.2.2 User Request Mapping and Execution

To execute the discovered user interaction not only on the migrated application, it has to be mirrored such that it can also be executed on the original application.

Due to the abstractions done in the reverse engineering step and the specializations and refinements performed in the forward engineering step, a simple mapping is not possible. Every change on the PSM and PIM level must be considered to allow for an efficient mapping of which original component corresponds to which migrated component and more importantly how the input for the methods must look like. Some migration processes might already produce the required transformation traces. In the context of the ARTIST project, we have defined a generic migration trace metamodel. The models conforming to this metamodel keep the migration trace representing the changes applied to the original application resulting in the migrated application. Using these artifacts, the migrated user request is mapped to one or more original user requests as soon as original components are found on the original application that correspond to the migrated components. As of

BY-SA

now, the prototype already supports a one-to-many mapping, i.e., a single migrated user request can be mapped to multiple original requests. After the original user requests have been built, both the migrated request and the original request(s) are sent to their respective target applications with the goal to be executed.

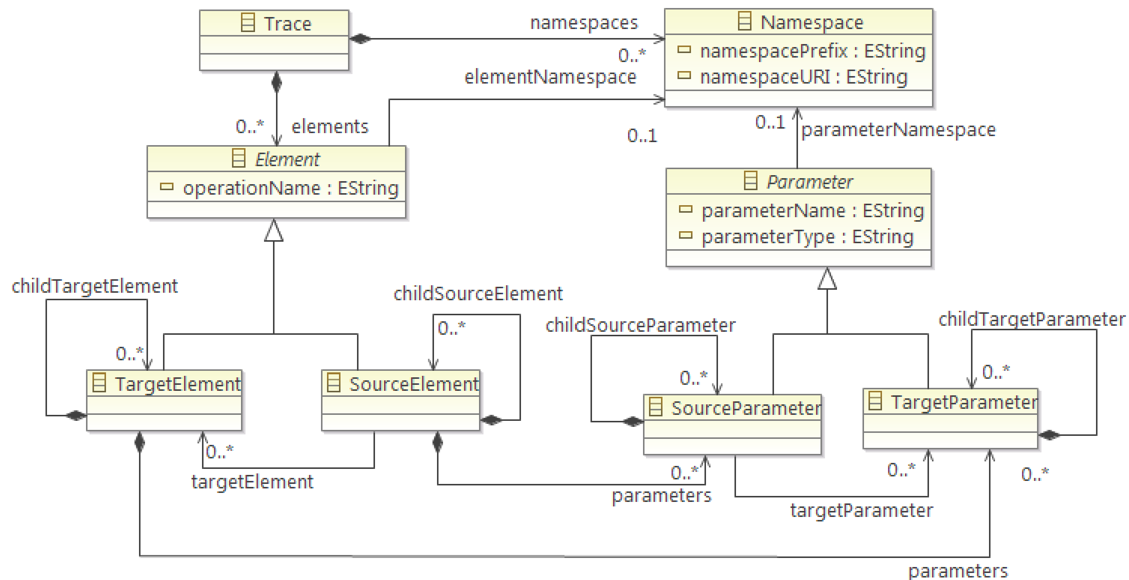


Figure 4: Web service migration trace meta model

3.2.3 Runtime Trace Extraction

After having applied equal requests to both the original application and migrated application, the output values produced by them can be extracted. In case of our prototype, the output produced represents a web service response containing, for example, information about application objects or database entries. Overall, the web service response represents the consolidated output produced by internal application activities. These internal activities might include method calls with their respective parameter values. Since our approach is categorized as gray-box testing, we do not look into the trace produced by the exact method call but into the overall output of the request. Hence, the web service response represents the consolidated output produced by the request that has been sent to the application.

However, in order to compare the output produced by both the original application and the migrated application, it needs to be extracted from the application. Consequently, retrieving the web service response and storing it into log files represents the Runtime Trace Extraction step.

3.2.4 Runtime Trace Abstraction

Immediately after the extraction has been completed, both application outputs are abstracted into an independent representation to allow further processing, such that their

BY-SA

comparison by the test oracle becomes feasible. In more detail, the extraction process only achieves a platform-specific representation of the user request output. To enable the comparison of both the output produced by the original application and the output produced by the migrated application, the individual platform-specific output has to be abstracted either into a platform independent level or a platform-specific compatible level. In the latter case, the platform-specific compatible level, the original application user request output (i.e., the original web service response) is abstracted into a forward compatible user request output and the migrated application user request output (i.e., the migrated web service response) is abstracted into a backward compatible user request output. Herewith, the possibility arises to compare both at their common platform-specific compatible level. In essence, the platform-specific compatible level represents an independent representation for any kind of information based on a specific platform or technology. In case of our prototype, the Simple Object Access Protocol (SOAP) has been chosen as the target technology to be used for evaluation purposes. Having transformed both application traces on the same level of abstraction (i.e., at platform-specific compatible level) allows their comparison, which is done in the next step.

3.2.5 Runtime Trace Comparison

The final step in comparing whether two applications behave the same given equal user requests is to compare the platform-specific compatible traces of both applications and yield a verdict. A positive verdict (pass) is yielded when both traces are equal in terms of output produced, and a negative verdict (fail) is yielded when the output is distinct. In the case of a negative verdict, the (first) element at fault is determined and reported back to the Testing Monitor. The comparison itself can, in case of SOAP web services, be made at different kind of levels. Within the prototype, SOAP web services can be compared at the following levels of comparison:

- Operation level (i.e., the operation name)
- Parameter level (i.e., the operation name and the parameter names)
- Parameter value level (i.e., the operation name, the parameter names, and the parameter values)

It is important to note that the output produced by a migrated request and its one or more corresponding original requests are compared before the next request is sent and executed.

3.3 Technical description

This section introduces the reader to the architecture of the end user-based testing prototype, a description on each of the architectural components and details about programming language, libraries, and so forth, which have been required to implement the prototype.

BY-SA

3.3.1 Prototype architecture

The architecture used to realize the methodology described earlier in this document is presented in this section. The architecture is divided into components in which each realizes a step from the methodology. An overview of the whole architecture is depicted in Figure 5, where a square represents a single component. Each component is described in its respective subsection with its necessary inputs and outputs and the task it accomplishes. While the architecture is described on a conceptual level, it has been realized on the Java platform that has no impact in case of SOAP web services as they are, at the lower-end, based on HTTP requests and responses. Hence, if the use case provider's application has been built upon Java or C# does not have an impact in the usage of the prototype.

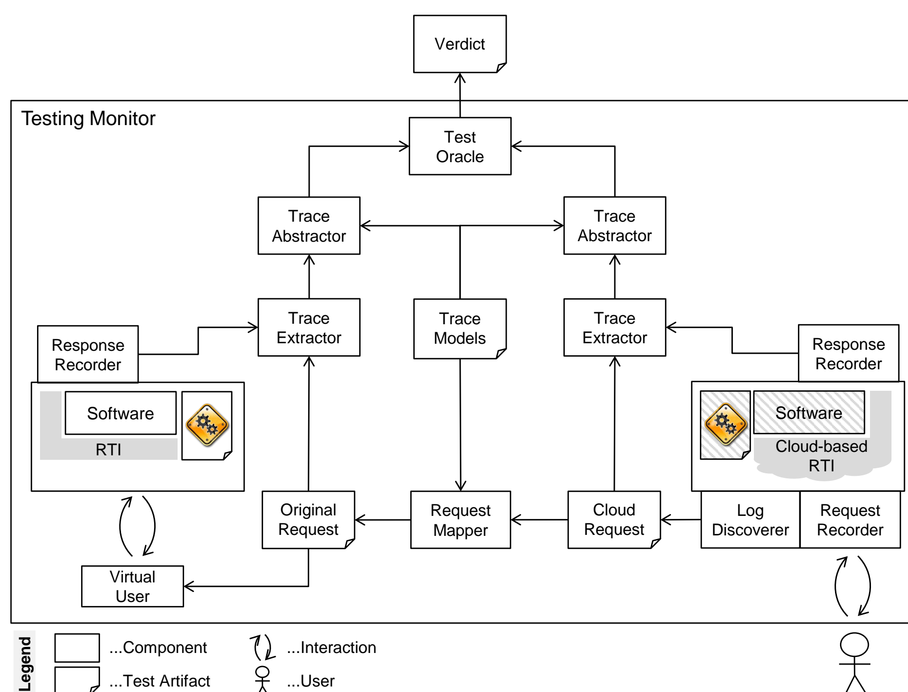


Figure 5: Architecture of end user-based testing

3.3.2 Testing Monitor

The Testing Monitor represents the base for realizing the end user-based testing methodology presented in the Section 3.2. It is responsible for the initialization of each sub component, the coordination between the components and the communication with the test developer. The Testing Monitor represents the proxy to which user requests are sent and it also has to assert that both, the migrated cloud application and the original application, are running and accessible. Each component managed by the Testing Monitor provides logs about its status and the output it produces. By using the Testing Monitor, other components produce logs that provide relevant information to the test developer. The Testing Monitor also has to assure that both, the migrated cloud application and the original application,

BY-SA

are running and accessible. The Testing Monitor collects component logs, processes them and presents relevant information to the test developer. In general, the Testing Monitor collects the application information, for example, the location of the original web service and the migrated web service, and initializes other components of the prototype.

3.3.3 User Request Discovery

As stated in Section 3.2.1 there are two possible ways to discover user requests. The first is to capture them directly when the user sends a request to the cloud application. The second is to use a logging mechanism of the migrated application to retrieve multiple user requests at once. While in the following both approaches are described, in the prototypical solution the latter approach (i.e., the request recorder) has been implemented. One of the reasons that led to that decision includes the fact that we want to avoid touching the source code of the migrated application as this would, for example, mean to cover several different platforms and any specific application in particular. Hence, platform independence, as it is the case with SOAP web services, would lose its benefits if any specific target application would have to be adapted to allow log discovery.

Log Discoverer. The task of the Log Discoverer is to retrieve the necessary log files from the migrated application and provide a sequence of requests for a specific user, i.e., a complete user session or user profile. As stated before, the ability to capture user requests in form of HTTP requests is often built into existing web servers which can act as load balancers for cloud applications. If it is also possible to retrieve all requests from specific Software-as-a-Service (SaaS) providers has yet to be determined. In those cases where the logging abilities of the web server or load balancer are not sufficient, logging can be implemented within the source code of the migrated application. For example, a SOAP request that is sent to a target application is not readable unless the target application directly issues a log-print statement. Hence, in order to allow customized logging, additional steps during forward engineering have to be taken. Furthermore, the source code logging mechanism might have a negative impact on the performance of the application (e.g., in case aspect oriented programming is used) and might have to be adapted every time the code itself changes. When taking requests from a log file, special attention has to be paid to keep the order of the requests chronologically as it is possible that requests depend on each other with respect to internal states. An advantage when using log files for test case generation is that clustering and prioritization techniques can be used to reduce the test suite size to reduce test effort while preserving the fault detection rate (for examples see Section 2).

To sum up, the Log Discoverer takes a log file as input and produces a sequence of user requests as output. The requests should stay in the chronological order in which they where performed by the user in case there are any dependencies between the requests.

Request Recorder and Response Recorder. The Request Recorder is responsible for capturing the requests directly from the user. The end user-based prototype incorporates

CC BY-SA

the Request Recorder and any request that is sent to the prototype first goes through the request recorder before any further actions, such as redirecting requests to the actual target application for execution, are performed. On the other hand, the Response Recorder first records any response that comes back from the target application to the end user-based prototype before any further processing is done. Since each request is forwarded directly by the prototype, no special attention has to be paid to the ordering of the requests. In summary, the Request Recorder handles the monitoring and recording of user interactions and forwards them to the respective components in the end user-based prototype for further processing.

3.3.4 User Request Mapper

The User Request Mapper is responsible for translating the request the user sent to the cloud application to a request that is understood by the original application with the goal to induce the same behavior on both applications. For converting a request from one application to the other, a mapping is needed. On the one hand the format of the request itself can be completely different, e.g., a HTTP request on the cloud side, but a direct method call or button click via script on the original side, on the other hand the structure or interfaces of the requested resources could have changed during the migration, e.g., the name of the responsible component or the number of parameters of a method. Thus, the changes made during the migration, i.e., the forward and reverse engineering phase, have to be considered. In the context of the ARTIST project, we have defined a generic migration metamodel. The models conforming to this metamodel keep the migration trace representing the changes applied to the original application resulting in the migrated application. These changes can be made available via so-called transformation trace or traces models. We have considered models conforming to the generic trace metamodel shown in Figure 4. It captures the original elements, their parameters, and how they relate to the migrated elements and their parameters.

In short, the User Request Mapper is a converter from migrated user requests to original user requests using trace models produced by the reverse and forward engineering steps in the migration. The resulting original user request is reported back to the Testing Monitor.

3.3.5 Virtual User

The Virtual User has the task to take a request for the original application and executing it. For the original application, it should make no difference whether the request comes from a real user or the Virtual User. Given the already transformed user request, the implementation in the exemplary case of SOAP is straight forward, i.e., a SOAP envelope containing the web service request to be sent to the target application. Hence, a Virtual User is a platform-specific user of a target platform.

BY-SA

3.3.6 Trace Extractor

After the real user and the Virtual User sent their requests, both the migrated application as well as the original application should have executed the same functional behavior. One way to represent this behavior is a runtime trace. One runtime trace conforms to exactly one execution of one user request and includes the web service operation call that the user request triggered and a possible output value (i.e., the web service response). We implemented an exemplary SOAP specific Trace Extractor that can be used in every ARTIST use case that uses a SOAP web service. Since the prototype can be executed externally to the use case application it is independent from the programming language or platform used by the use case. However, if a different technology (e.g., REST or even not a web service technology) is used, a different Trace Extractor implementation has to be built.

Summarized, the Trace Extractor retrieves the runtime trace for a specific request from a given application. The extraction technique and format of the trace depends on the technology used for the application, so multiple Trace Extractors have to be implemented. However, the exemplary Trace Extractor (i.e., the SOAP Trace Extractor) developed for the prototype is able to extract a SOAP trace from any SOAP web service call independent from the programming language or platform used to deploy the web service.

3.3.7 Trace Abstractor

Given the runtime traces extracted from the Trace Extractor, the Trace Abstractor must bring each trace into a platform-independent or platfrom-specific compatible format to be able to compare runtime traces in the next stage of the process. Similar to the Trace Extractor, a dedicated Trace Abstractor has to be written for every technology-specific runtime trace type. Within the exemplary solution for the prototype we developed a SOAP specific Trace Abstractor that is able to abstract a SOAP trace (i.e., a SOAP response envelope). To be more specific, both the migrated SOAP response and the original SOAP response are brought into a platform-specific compatible level. This means that the migrated SOAP response is abstracted such that it is backward compatible to the original system and the original SOAP response is abstracted such that it is forward compatible to the migrated system. To achieve this, we use the migration trace model that provides information on how a original web service operation and their parameters correspond to a migrated web service operation and their parameters, respectively.

In summary the Trace Abstractor is responsible for the final step before the runtime traces can be compared. One runtime trace extracted from a specific application is abstracted to a platform-specific compatible level using the migration trace model.

BY-SA

3.3.8 Test Oracle

The final step in realizing the methodology presented in Section 3.2 is to compare two platform-specific compatible runtime traces that origin from the same user request. Since both runtime traces are at a high level of abstraction, their comparison is straight forward. After the comparison, the Test Oracle yields a verdict of either *pass* if the comparison was successful and the two runtime traces are similar according to the criteria specified in Section 3.2.5, or *fail* if the two runtime traces represent different application behavior. A comparison is considered successful if both traces produce the same output.

3.4 Technical specification

The end user-based testing prototype has been developed based on the Java platform. The build management tool chosen for the prototype is Apache Maven and the following dependencies have been used within the context of the prototype:

- JUnit – for unit testing support
- Apache Commons IO – for file input/output support
- Apache Commons Collections 4 – for multi map support
- com.sun.xml.txw2 – for indenting XML stream writer support
- Apache Axis 2 – for SOAP support, the successor to the Apache Axis SOAP stack
- Eclipse EMF (Eclipse Modeling Framework) – for modeling support in handling the migration trace models
- Google App Engine API – Google App Engine support for the Java Runtime Environment
- JavaX Servlet API – Java servlet container support, part of Java Enterprise Edition

In the next section, the delivery and usage of the prototypical implementation is mentioned.

BY-SA

4 Delivery and Usage

This section describes the delivery and usage of the ARTIST end user-based prototype in terms of package information, installation instructions, user manual, licensing information, and information where to retrieve the source code.

4.1 Package Information

The ARTIST end user-based prototype is delivered as a Maven parent project containing four modules. While the parent project is named "eu.artist.postmigration.eubt", the modules are referred to and described as follows.

4.1.1 Prototype Module

The "eu.artist.postmigration.eubt.prototype" module contains the main functionality of the prototype. Hence, the components described in Section 3.2 are all contained in this module. The "src/test/java" part of this module contains the class SOAPTestCase containing tests on the setup procedure of the Monitor, the creation of a request targeted to the modernized application and its following execution and response evaluation.

4.1.2 Migration Trace Model Module

The "eu.artist.postmigration.eubt.migrationtracemodel" has been built following the conclusion that a trace model is necessary to be able to build backward compatible requests from the migrated request. Hence, the trace meta model contained in this module provides the possibility to build models containing information on how a specific source element and its source parameters correspond to one or more target elements and their target parameters, respectively.

4.1.3 Exemplary Migrated Web Service Module

The module "eu.artist.postmigration.eubt.example.ws.migrated" represents an exemplary SOAP web service implementation that can be deployed on the Google App Engine. The web service is built on top of the JAX-WS (Java API for XML Web Services) reference implementation. The example itself has been built with the semantics of the Petstore application in mind. Within this module, changes as imposed by the migration process have already been realized. Hence, "eu.artist.postmigration.eubt.example.ws.migrated" represents the exemplary migrated web service.

BY-SA

4.1.4 Exemplary original Web Service Module

Like the module "eu.artist.postmigration.eubt.example.ws.migrated", the "eu.artist.postmigration.eubt.example.ws.original" module contains an example web service based on the Petstore application. However, this module represents the original web service and accordingly does not yet contain changes as imposed by the migration process. In other words, this module represents the exemplary original web service.

4.2 Installation Instructions

As listed in Section 4.7, and other than the source code of the ARTIST end user-based prototype itself, the target machine on which the test cases are intended to be executed, requires the build automation software Apache Maven[2] to be installed either in form of a native installation or in form of an Eclipse IDE (integrated development environment) plugin. If this requirement is fulfilled, the ARTIST end user-based prototype can be built and the prepared test cases can be executed as described in Section 4.3. Furthermore, to use different web services and build own test cases, the Eclipse Platform (Modeling Edition) and Eclipse EMF is required to be installed.

4.3 User Manual

Within the scope of this section, both the creation of ARTIST end user-based test cases and the execution of these test cases are explained.

4.4 Run Test Cases

Initially, the source code needs to be checked out from the ARTIST public Github repository. In order to build the source code and execute existing test cases, it can either be imported into the workspace of an IDE such as Eclipse or from the command line. It is important to note that both procedures require the build automation tool Apache Maven to be installed on the target machine. The latter tool can either be installed natively and/or as an Eclipse plugin. However, if it is installed natively (only), then the build process can only be started from the command line and if an Eclipse Maven Integration plugin (e.g., "m2e – Maven Integration for Eclipse") is used, it can be started from inside the Eclipse IDE. To display the migration trace meta model and models either the Eclipse Modelling distribution or another Eclipse distribution with manually installed Eclipse EMF plugin is required.

To ease the usage of the ARTIST end user-based testing prototype, both exemplary web services (i.e., the migrated web service and the original web service) have been deployed on the Google App Engine. Thus, none of the exemplary web services needs to be deployed before running the exemplary test cases. From a functional perspective it does not

[2] Apache Maven can be retrieved online at http://maven.apache.org/download.cgi.

BY-SA

make a difference if a web service is deployed on a local machine or any another server including the Google App Engine or any other cloud infrastructure.

Optionally: If the exemplary web services are desired to be run locally instead of using the already deployed web services running on the Google App Engine, the `SOAPTestCase` class that resides in the "`eu.artist.postmigration.eubt.prototype`" module needs to be adapted to match the correct URLs pointing to the locally or elsewhere running web service. To startup the local web service, the user switches to the exemplary web service project folder (i.e., "`eu.artist.postmigration.eubt.example.ws.migrated`" or "`eu.artist.postmigration.eubt.example.ws.original`") inside the console and types the following command:

```
> mvn appengine:devserver
```

Finally, to run the build process and any associated test cases in the command line the following command is issued inside the parent project folder "`eu.artist.postmigration.eubt`":

```
> mvn clean install
```

Issuing the above command executes the build process and any associated test cases. In case the final reactor summary shows "`BUILD SUCCESS`", the build process including any associated test cases have been executed successfully. In case of a test case failure or any other build related failure will immediately be displayed by a final reactor summary "`BUILD FAILED`".

Alternatively: The entire build process and the execution of individual test cases can also be issued from inside the Eclipse IDE. To start the build process, right click on the main project (i.e., "`eu.artist.postmigration.eubt`") and select "`Run As > Maven install`" (cf. Figure 6). The JUnit test cases inside the SOAPTestCase class can be executed by right clicking on the class file inside the Eclipse Project Explorer and selecting "`Run As > JUnit Test`" (cf. Figure 7).

CC BY-SA

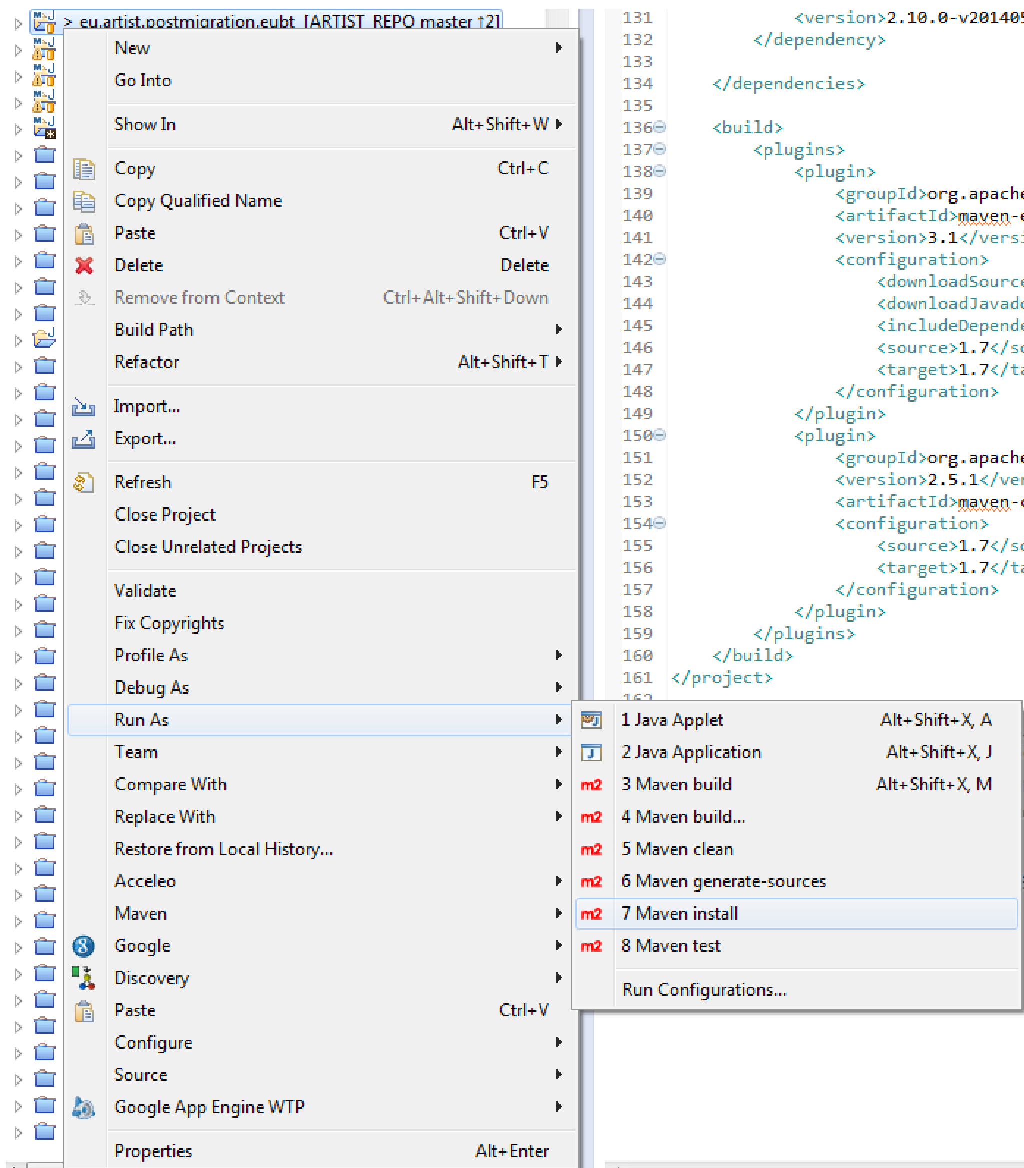


Figure 6: Running the Maven install inside Eclipse IDE.

BY-SA

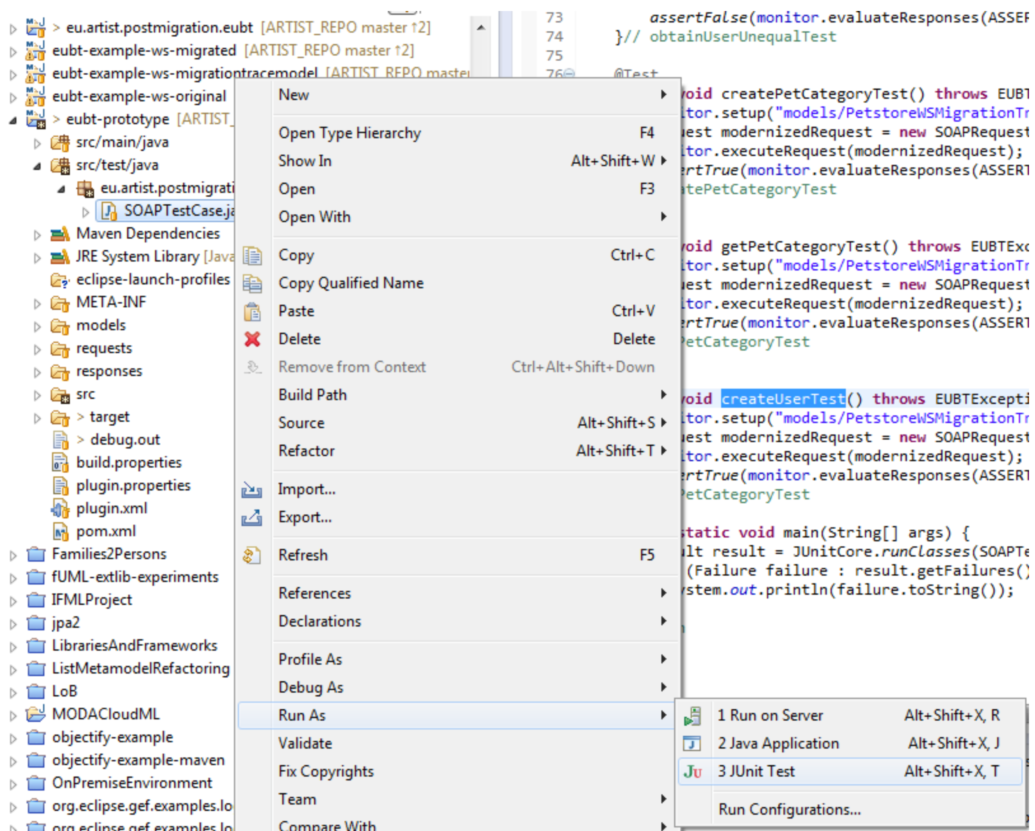


Figure 7: Running the JUnit tests inside Eclipse IDE.

4.5 Build Test Cases

The JUnit test or other source code example to that represents an entire ARTIST end user-based test case has the following minimum required steps. In the following we go through a list of steps that are required to build up such a test case.

Step 1: Setup the migrated application and original application information.

```
1 SOAPApplication modernizedApplication = new
    SOAPApplication(APPLICATION_TYPE.MODERNIZED_APPLICATION,
    MIGRATED_APP_SERVICE_URL, MIGRATED_APP_WSDL,
    MIGRATED_APP_WSDL_SCHEMA, "WebservicesDemoService",
    "WebservicesDemoPort", ORIGINAL_APP_SERVICE_NAMESPACE);

3 SOAPApplication originalApplication = new
    SOAPApplication(APPLICATION_TYPE.ORIGINAL_APPLICATION,
    ORIGINAL_APP_SERVICE_URL, ORIGINAL_APP_WSDL,
    ORGINAL_APP_WSDL_SCHEMA, "WebservicesDemoService",
    "WebservicesDemoPort", MIGRATED_APP_SERVICE_NAMESPACE);
```

Step 2: Setup the monitor component.

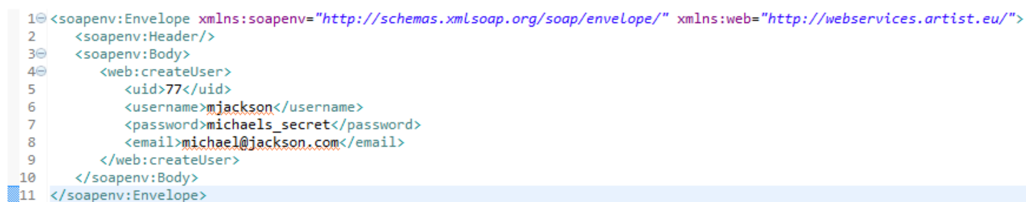
BY-SA

```
Monitor monitor = new SOAPMonitor(modernizedApplication,
    originalApplication);
monitor.setup("models/PetstoreWSMigrationTrace_Request.xmi",
    "models/PetstoreWSMigrationTrace_Response.xmi");
```

Note that the "`PetstoreWSMigrationTrace_Request.xmi`" and "`PetstoreWSMigrationTrace_Response.xmi`" files in step 2 refer to the request trace model and the response trace model, respectively. These models enclose information on the correspondence between original elements and migrated elements. Within the Eclipse IDE, the "Sample Reflective Ecore Modeling Editor" has been used to create the aforementioned model files.

Step 3: Setup the migrated application and original application information. The XML file location stated in step 3 refers to a XML file containing a concrete migrated SOAP envelope in XML notation. Figure 8 illustrates the exemplary SOAP envelope used in step 3. This exemplary SOAP envelope, which is loaded from the file system, represents a captured migrated SOAP request. Such a request could be captured using functionality of capture and replay tools or similar. However, for our implementation we assume that such SOAP requests already exist in the file system. The corresponding original SOAP envelope, to be issued to the original web service, is created by the prototype using the supplied migration trace model.

```
Request modernizedRequest = new
    SOAPRequest(APPLICATION_TYPE.MODERNIZED_APPLICATION,
    PATH_TO_MIGRATED_REQUEST_FOLDER + "createUser.xml",
    "http://webservices.artist.eu/CreateUser");
```



```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:web="http://webservices.artist.eu/">
   <soapenv:Header/>
   <soapenv:Body>
      <web:createUser>
         <uid>77</uid>
         <username>mjackson</username>
         <password>michaels_secret</password>
         <email>michael@jackson.com</email>
      </web:createUser>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 8: Exemplary SOAP request envelope referenced in step 3.

Step 4: Specifying the modernized request to be issued. The migrated user request, as loaded from the file system in the previous step, is executed through the monitor component of the prototype.

```
monitor.executeRequest(modernizedRequest);
```

Step 5: Evaluating the response retrieved from the web service.

```
assertTrue( monitor.evaluateResponses(
    ASSERTION_LEVEL.ATTRIBUTE_VALUE_LEVEL ) );
```


CC BY-SA

The evaluation in step 5 can be made at different levels of comparison as listed in Section 3.2.5. The responses retrieved by the Monitor and (internally) evaluated by the Test Oracle will return “`true`” if the verdict produced is “`PASS`” and “`false`” in case the verdict produced is “`FAIL`”.

Analyzing the output.

Listing 1 depicts the first part of the output produced by executing the “createUser” test case. Lines 1 to 4 highlight that both the Web Service Description Language (WSDL) file and its corresponding schema file for the modernized application and the original application have been successfully found at their specified locations. While line 5 details that the migrated user request stored in file “createUser.xml” has been successfully loaded into the prototype, line 8 states that a corresponding original user request has been created and stored in an equal named file located in another subfolder. The next output, line 10, refers to the same file as in line 8, but this time the original user request is loaded from the file which concludes the work by the “SOAPUserRequestMapper” as depicted by line 12. At this point in time, both the migrated user request and its corresponding original user request(s) are ready to be fired to their parallel running web services. Lines 14 to 15 acknowledge the success of the latter user request firing.

Listing 2 shows the prototype output produced after both the migrated user request and its corresponding original user request(s) have been fired to their respective web service applications. The first action taken with the received SOAP response is to check if it contains a valid body and hence no error has been returned (cf. line 1). Then, the received SOAP response is abstracted into a platform-specific compatible format by the `SOAPTraceAbstractor` (cf. line 3). Line 4 loads the previously stored migrated user response file to be further used by the prototype. Lines 6 to 10 are corresponding to the previous lines in Listing 2 but for the original user response. At this point in time, both SOAP responses have been brought to platform-specific compatible level (i.e., transformed into an `IndendentSOAPTrace`) such that they can be compared. Lines 12 to 13 validate both application responses against their respective schema. Line 15 to 19 show that equal operation names, element names, element values, and attribute values have been found in the user responses. With line 21 the prototype concludes by having the `SOAPTestOracle` produce a verdict with value “`pass`” based on a comparison at attribute value level (see levels of comparison in Section 3.2.5).

CC BY-SA

```
[SOAPApplication] Successfully found MODERNIZED_APPLICATION service WSDL at http://artist-jaxws-migrated.appspot.com/WebservicesDemoService.wsdl
[SOAPApplication] Successfully found MODERNIZED_APPLICATION service Schema at
    http://artist-jaxws-migrated.appspot.com/WebservicesDemoService_schema1.xsd
[SOAPApplication] Successfully found ORIGINAL_APPLICATION service WSDL at http://artist-jaxws-original.appspot.com/WebservicesDemoService.wsdl
[SOAPApplication] Successfully found ORIGINAL_APPLICATION service Schema at
    http://artist-jaxws-original.appspot.com/WebservicesDemoService_schema1.xsd

[SOAPRequest] Successfully created SOAP request from file at C:\ARTIST_REPO\02_development\End User Based
    Testing\eu.artist.postmigration.eubt.prototype\requests\migrated\createUser.xml

[SOAPUserRequestMapper] Successfully stored mapped SOAP envelope to file at C:\ARTIST_REPO\02_development\End User Based
    Testing\eu.artist.postmigration.eubt.prototype\requests\original\createUser.xml

[SOAPRequest] Successfully created SOAP request from file at C:\ARTIST_REPO\02_development\End User Based
    Testing\eu.artist.postmigration.eubt.prototype\requests\original\createUser.xml

[SOAPUserRequestMapper] Successfully mapped source SOAP request to target SOAP request(s).

[SOAPApplicationUser] Fired request action http://webservices.artist.eu/CreateUser to location
    http://artist-jaxws-migrated.appspot.com/WebservicesDemoService
[SOAPApplicationUser] Fired request action http://webservices.artist.eu/CreateUser to location
    http://artist-jaxws-original.appspot.com/WebservicesDemoService
```

Listing 1: Output produced by executing the createUser test case (part 1).

BY-SA

```
[SOAPTraceAbstractor] Successfully validated SOAP body content <ns3:createUserResponse
    xmlns:ns3="http://webservices.artist.eu/"><return>true</return></ns3:createUserResponse>
[SOAPTraceAbstractor] Successfully stored abstracted SOAP envelope to file at C:\ARTIST_REPO\02_development\End User Based
    Testing\eu.artist.postmigration.eubt.prototype\responses\PIM\migrated\createUserResponse.xml

[SOAPResponse] Successfully created SOAP response from file at C:\ARTIST_REPO\02_development\End User Based
    Testing\eu.artist.postmigration.eubt.prototype\responses\PIM\migrated\createUserResponse.xml

[SOAPTraceAbstractor] Successfully validated SOAP body content <ns3:createUserResponse
    xmlns:ns3="http://webservices.artist.eu/"><return>true</return></ns3:createUserResponse>
[SOAPTraceAbstractor] Successfully abstracted source SOAP response to target SOAP response.
[SOAPTraceAbstractor] Successfully stored abstracted SOAP envelope to file at C:\ARTIST_REPO\02_development\End User Based
    Testing\eu.artist.postmigration.eubt.prototype\responses\PIM\original\createUserResponse.xml

[SOAPResponse] Successfully created SOAP response from file at C:\ARTIST_REPO\02_development\End User Based
    Testing\eu.artist.postmigration.eubt.prototype\responses\PIM\original\createUserResponse.xml

[SOAPTraceAbstractor] Successfully validated full MODERNIZED_APPLICATION response trace against its schema.
[SOAPTraceAbstractor] Successfully validated full ORIGINAL_APPLICATION response trace against its schema.

[IndependentSOAPTrace] Trace comparison: found equal operation names ’createUserResponse’.
[IndependentSOAPTrace] Trace comparison: found equal element names ’createUserResponse’.
[IndependentSOAPTrace] Trace comparison: found equal element names ’return’.
[IndependentSOAPTrace] Trace comparison: found equal ’return’ element values ’true’.
[IndependentSOAPTrace] Trace comparison: all attribute values are equal.

[SOAPTestOracle] VERDICT based on a assertion at ’ATTRIBUTE_VALUE_LEVEL’: PASS.
```

Listing 2: Output produced by executing the createUser test case (part 2).

BY-SA

4.6 Licensing Information

The ARTIST end user-based prototype is made available as an open-source solution on the ARTIST public Github repository. The selected license is the Eclipse Public License (EPL) [3] that is known as a “commercially-friendly” open source license. The ARTIST end user-based prototype is provided under the EPL.

4.7 Download

The end user-based testing prototype minimum requirement is to have Apache Maven installed on the testing machine. Thus, having Apache Maven installed, the prototype and any prepared tests can already be compiled and executed using the “`mvn clean install`” command. In order to build own test cases on other SOAP web services or to extend the implementation for other end user-based test scenarios, the following software is needed:

- Java Development Kit (JDK) Version 1.7
- Eclipse Platform (Modeling Edition)
- Apache Maven (required to execute the build process) or Eclipse Maven Integration plugin (if the build process is desired to be started from inside the Eclipse IDE)
- Eclipse EMF (to create trace meta models and models)

The source code is available in the ARTIST public GitHub repository [4].

[3] Eclipse Public License - http://www.eclipse.org/legal/epl-v10.html

[4] The ARTIST GitHub repository is available online at https://github.com/artist-project/ARTIST/tree/master/source/Tooling/post-migration/eubt.

BY-SA

5 Conclusion

In this deliverable, we presented and demonstrated the first version of the end user-based testing prototype which is developed in the context of ARTIST. We introduced the migration trace meta model which aims to represent the trace produced by migrated an original system to a modernized system. To render such end user-based tests more concrete and specific to a particular web service, we developed exemplary web services and created associated tests for them in order to compare their behavior. Based on the created exemplary web services, that are based on the Petstore semantics, we demonstrated the use of the end user-based testing prototype. Hence, we described each component of the methodology and its part in the final goal to evaluate the behavioral equivalence between the migrated system and the original system.

The main focus of this deliverable was on highlighting the feasibility of the methodology presented in WD11.2, and for this we developed a prototype capable of behavioral equivalence evaluation between two different web services based on the SOAP technology.

In summary, the methodology is as follows. User requests are assumed to be created by a real user on the migrated software and stored to the file system. The prototype then uses these user request files and a migration trace model to map the migrated user request to one or multiple original user requests to induce equivalent behavior on both sides. To verify whether the two systems actually behave equally, the respective runtime traces are extracted from both system and abstracted into a platform-specific compatible trace. After comparing both traces, which are compatible to each other, a verdict about their equivalence can be yielded. We define a runtime trace as the output of a request (i.e., the response), operation calls, their parameters, as well as their parameter values. For the comparison we build a test oracle that can distinguish a comparison to be successful if both traces contain the response of equivalent operation calls, or equivalent operation calls and their parameter names, or equivalent operation calls, their parameter names, and their parameter values.

To realize this methodology, a component-based architecture has been introduced. The presented architecture shows how the methodology has been implemented using the Java technology. Each step of the methodology is implemented by one or more components and one controller (i.e., the monitor) takes care of their execution and coordination. The same controller is also responsible for communicating with the tester, i.e., showing errors and giving feedback. Since the prototype can be executed externally to the use case and is, hence, not dependent on the use case platform such as .NET or Java, it supports SOAP web services provided by virtually any underlying platform.

However, in order to support user requests issued by different web services, such as, for example, REST, or even UI-based applications, different implementations have to be realized. Yet, such implementations can build on the same methodology and similar architecture as the SOAP based prototype. Thus, future work can include to build the same methodology for several different technologies, to replace the migration trace model with

CC BY-SA

the actual migration trace produced during the ARTIST migration process, and to transform the platform-specific compatible trace abstraction to a platform independent trace abstraction such as provided by UML models. To conclude, the end user based testing methodology already presented in this document is feasible in case of web services and in particular in case of the SOAP technology.

CC BY-SA

6 References

[1] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing Web applications by modeling with FSMs. *Software & Systems Modeling*, 4:326–345, 2005.

[2] Batini, C. and Lenzerini, M. and Navathe, S.B. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys (CSUR)*, 18(4):323–364, 1986.

[3] Giuseppe A. Di Lucca and Anna Rita Fasolino. Testing web-based applications: The state of the art and future trends. *Information and Software Technology*, 48(12):1172–1186, 2006.

[4] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, 2005.

[5] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 49–59, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Robert Godin, Rokia Missaoui, and Hassan Alaoui. Incremental concept formation algorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.

[7] James H. Hicinbothom and Wayne W. Zachary. A Tool for Automatically Generating Transcripts of Human-Computer Interaction. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 37, page 1042. Sage Publications, 1993.

[8] J. Simpson and E. Weiner. *The Oxford English Dictionary*. Oxford University Press, 1989.

[9] Jin-hua Li, TIAN Hengxiang, and Dandan Xing. Clustering user session data for web applications test. *Journal of Computational Information Systems*, 7(9):3174–3181, 2011.

[10] Jin-hua Li and Dan-dan Xing. User session data based web applications test with cluster analysis. In *Advanced Research on Computer Science and Information Engineering*, volume 152 of *Communications in Computer and Information Science*, pages 415–421. Springer Berlin Heidelberg, 2011.

[11] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. An object-based data flow testing approach for web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(02):157–179, 2001.

[12] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program exe-

BY-SA

cutions. *SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[13] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[14] S. Sampath, R.C. Bryce, G. Viswanath, V. Kandimalla, and A.G. Koru. Prioritizing user-session-based test cases for web applications testing. In *1st International Conference on Software Testing, Verification, and Validation*, pages 141 –150, 2008.

[15] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, ASE '04, pages 132–141. IEEE Computer Society, 2004.

[16] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 253–262. ACM, 2005.

[17] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. *SIGSOFT Software Engineering Notes*, 25(5):158–167, 2000.

[18] Tsichritzis, D. The equivalence problem of simple programs. *Journal of the ACM (JACM)*, 17(4):729–738, 1970.

CC BY-SA