A Framework of Testing as a Service

Lian Yu, Le Zhang, Huiru Xiang, Yu Su, †Wei Zhao, †Jun Zhu
School of Software and Electronics, Peking University, Beijing, 102600, PRC
†China Research Center, IBM, Beijing, 100094, PRC

Abstract

This paper presents a framework of Testing as a Service (TaaS) which a new model to improve the efficiency of software quality assurance. We exploit the related key technical issues, and build a reference architecture of TaaS based on ontology, process automation and SOA techniques. A prototype for unit testing is implemented to demonstrate the validity of the framework.

Keywords: Testing As a Service (TaaS), Test Case Generation, Test Environment, Reference Architecture.

1. Introduction

Software testing is a fundamental activity of software quality assurance. Many tools are developed to assist developers and testers with their testing. Learning and writing test cases using those tools usually taxes much effort of software developers.

TaaS (Testing as a service) provides the tenants with testing services, which allow them to submit their test tasks, such as auto-generation of test cases and test auto-execution, and get desired test results from TaaS platform. Three key technical aspects of TaaS are needed to deal with. One is to match tenant requests with platform capability intelligently, so that tenants do not need to care about which exact testing services will be invoked by the platform. The second is to explore techniques to decompose and compose web services to make TaaS extendable. The third is to allow community developers to contribute to the platform by publishing their testing services and deploying them to the platform.

OASIS UDDI is an XML-based registry of web services [2] ; however, it just provides simple endpoint lookups and does not support auto-service matching. It also does not manage testing lifecycle. IBM WSRR (WebSphere Service Registry and Repository) [3] has a powerful capability to select, invoke, govern, and reuse services via lifecycle management, and it is suitable for enterprise development. TaaS platform needs a compact light-weighted capability for testing service matching, composting and publishing. The paper addresses the following three issues:

1) Building an ontology model to intelligently match test tasks that a tenant requests, and test capability that TaaS platform provides, and employ SWRL (Semantic Web Rule Language) [5] to represent the test service matching rules. MapReduce approach is used to sort tasks from different tenants.
2) Decomposing and composting testing services to fulfill tenant's test requirements. Once matching rule engine draws out a set of testing services, a compositor orchestrates those testing services in proper logical order to accomplish tenant's testing tasks.
3) Supporting testing service contributors to publish their testing services to TaaS platform. The contributors provide their service meta-data as ontology individuals that are used for matching and reasoning.

The rest of the paper is organized as follows. Section 2 illustrates the architecture of the TaaS platform. Section 3 constructs an ontology testing model and presents matching rules. Section 4 describes the process of matching and scheduling. Section 5 presents testing service composition. Section 6 illustrates experiments to evaluate the TaaS platform. Finally, Section 7 concludes the paper and outlines the future work.

2. Architecture of Testing as a Service

Figure 1 shows the architecture of testing as a service, which consists of four layers: Test Service Tenant and Contributor layer, Testing Service Bus layer, Testing Service Composite layer, and Testing Service Pool layer.

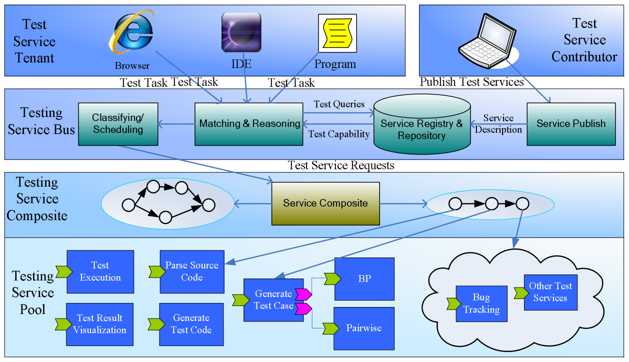

Figure 1: Architecture of Testing as a Service

- The top layer has two parts, supporting test service tenants and contributors to interact with TaaS. The tenants of TaaS can use a portal, or an integrated development environment (IDE) to access testing

services via Internet, even employ a programmatic approach to interact with the TaaS platform. Test service contributors publish and deploy their implemented services to TaaS platform.
- The second layer is the testing service bus (TSB), which stays between end users and testing services, and consists of four core components: Matching and Reasoning, Classifying and Scheduling, Service Registry and Repository, and Service Publish.
- The third layer is the test service composite, which orchestrates a set of test services and invokes proper test services in a proper order.
- The forth layer is the test service pool, which hosts an array of test services for different types of unit test tasks, including test case generation service, test execution service, and result presentation service.

3. Ontology Modeling

This section uses ontology techniques to model software testing-related concepts and their relations, and uses SWRL to model the rules matching tenants' test tasks with TaaS test capabilities.

Test tasks of tenants are modeled as an OWL class, *TestTask*, which contains OWL sub-classes of *TargetUnderTest*, *TestResource*, *TaskContraints*, and *TestType*, and has a *TestActivity* and *TestSchedule* as shown in Figure 2.

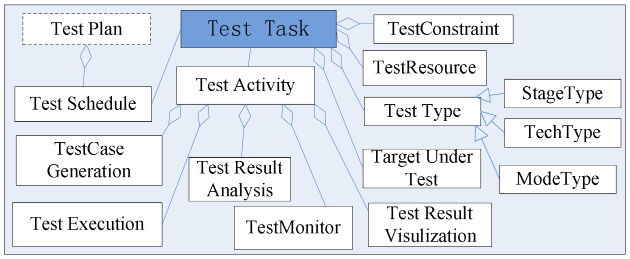

Figure 2: TestTask related concepts and relations

TestResource shows the requirements of hardware and software to enable a test task, for example, a computer to generate test cases and dependent software package. *TestConstraint* specifies test constrains to meet, such as path coverage or condition coverage. *TestType* can be categorized in terms of development stages (e.g., unit testing, integration testing or system testing), techniques (white-box or black-box) and test mode (e.g. static test or dynamic test). *TestActivity* includes *TestCaseGeneration*, *TestExecution*, *TestMonitor*, *TestResutlAnalysis* and *TestVisualization*. Each OWL class has associated properties to indicate its attributes and specify the connections with other OWL classes.

Test capability of TaaS is modeled as an OWL class, *TestCapability*, which consists of several OWL classes as shown in Figure 3. OWL class *TestingService* represents the testing-related services of TaaS platform, and has several subclasses to match each testing activity in *TestTask*. OWL class *QualityOfService* indicates the competence to meet test schedule and the degree of tenant satisfaction. If there is no capability that matches a test task from a tenant, the request will be forwarded to *Tester*, which can be a human being or a team and takes proper measure to handle the situation.

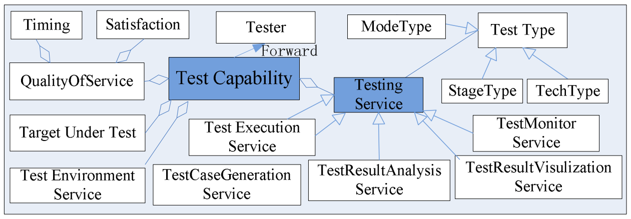

Figure 3: TestCapability related concepts and relations

We define nine rules using SWRL to match test tasks of tenants with test capability of TaaS. Figure 4 shows the rule of matching *TestCaseGenerationService* with *TestTask*. The SWRL rule reads as "If the test types of the test task are the same as one of *TestCaseGeneration* Services, and features of the *TestCaseGeneration* Service satisfy the task requirement, then match the test task with the *TestCaseGeneration* Service".

```
......
TestCaseGeneration_Service(?gen_service)  ∧
swrlb:stringEqualIgnoreCase(?tt_techType, ?tc_techType)∧
swrlb:stringEqualIgnoreCase(?tt_stageName, ?tc_stageName)∧
swrlb:stringEqualIgnoreCase(?tt_modeType , ?tc_modeType) ∧
swrlb:stringEqualIgnoreCase(?tt_Coverage,?tc_Coverage)  ∧
swrlb:stringEqualIgnoreCase(?tt_loop, ?tc_loop)  ∧
swrlb:stringEqualIgnoreCase(?tt_strategy, ?tc_strategy)  ∧
swrlb:stringEqualIgnoreCase(?tt_paraInput, ?tc_paraInput)
 → matchForTestCaseGenerationService(?test_task, ?gen_service)
```

Figure 4: Match TestCaseGenerationService

4. Matching and Scheduling

Upon arrival of a tenant request, reasoning service on TSB extracts the values of *TestTask*, utilizes Protégé [6] APIs to set up the corresponding individuals, sends those individuals and pre-defined SWRL rules to Jess engine [7] through Protégé Bridge, and gets matching results back as shown in Figure 5.

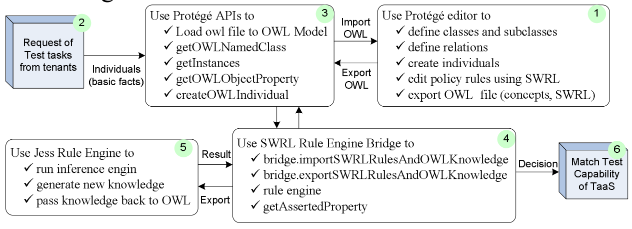
2
Request of Test tasks from tenants
Individuals (basic facts)
3
Use Protégé APIs to
✓ Load owl file to OWL Model
✓ getOWLNamedClass
✓ getInstances
✓ getOWLObjectProperty
✓ createOWLIndividual
Import OWL
Export OWL
1
Use Protégé editor to
✓ define classes and subclasses
✓ define relations
✓ create individuals
✓ edit policy rules using SWRL
✓ export OWL file (concepts, SWRL)
5
Use Jess Rule Engine to
✓ run inference engin
✓ generate new knowledge
✓ pass knowledge back to OWL
Result
Export
4
Use SWRL Rule Engine Bridge to
✓ bridge.importSWRLRulesAndOWLKnowledge
✓ bridge.exportSWRLRulesAndOWLKnowledge
✓ rule engine
✓ getAssertedProperty
Decision
6
Match Test Capability of TaaS

Figure 5: Reasoning Process

Classifier sorts requests from different tenants into task classes, and Scheduler assigns each class of test activities to computers to perform corresponding test activities, as shown in Figure 6. Map phase performs concurrently test case generation and/or test execution, and Reduce phase acts upon test result analysis and generates test reports back to each tenant.

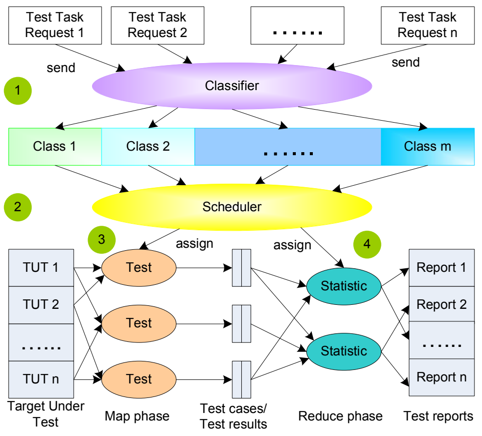

Figure 6: Classifying and Scheduling

5. Service Decomposition and Composition

Decomposition of test services is a process to identify, specify and realize testing services as shown in Figure 7. We start from inspecting testing process and identifying each step for possible service candidates in terms of technical feasibility and testing domain necessity, and then specify identified services in terms of interface, input and output, finally implement the testing services. Implementing services for Unit testing has three approaches in this research: 1) develop from scratch; 2) wrap existing test functionality; and 3) integrate basic testing services. The process is incremental and iterative.

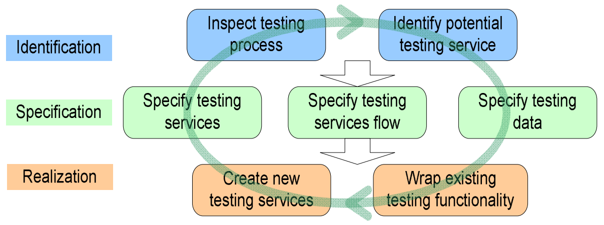
Identification
Specification
Realization
Inspect testing process
Identify potential testing service
Specify testing services
Specify testing services flow
Specify testing data
Create new testing services
Wrap existing testing functionality

Figure 7: Process of Developing Testing Services

The goal of service composition is to determine which service providers can collaborate, and then to derive new services out of existing ones. Individual test service need to be orchestrated in proper order and invoked to achieve the tenant's test task. BPEL (Business Process Execution Language) [12] is a typical industry solution for service composition and needs a heavy-weighted container. A test process typically consists of planning, test case generation, test execution and test result analysis. We use the dependency injection pattern [13] to configure the process template pre-defined. TaaS Composite service infers tenant's intents and determines which step(s) should be involved, then automatically instantiates the process.

6. Experiments

The realization of TaaS platform is built on Apache Tuscany, which simplifies the task of developing SOA solutions by providing a comprehensive infrastructure for SOA development and management that is based on SCA (Service Component Architecture) and SDO (Service Data Object) [14] standards. It's implemented in Java and C++. We use the Java version 1.3.2 to build our solution. The experiment is run on two PC machines connected in LAN.

The first experiment compares the service invocation times in local and remote environments. We select two white box test case generation services: BP (Basis Path) and BRO (Branch and Relational Operator) [9] . BP Testing derives a basis set, including linearly independent paths that can be used to construct any path through the program flow graph [15] . BRO technique guarantees the detection of branch and relational operator errors in a condition provided that all Boolean variables and relational operators in the condition occur only once and have no common variables.

We also choose anther two black box test case generation services: GEO (Generalized Extremal Optimization) and Pairwise. GEO algorithm which belongs to Evolutionary algorithm was used to generate test data automatically of path testing [16] . Pairwise testing is one combinatorial strategy in which every possible pair of the values of any two test input parameters is covered by at least one test case [17] .

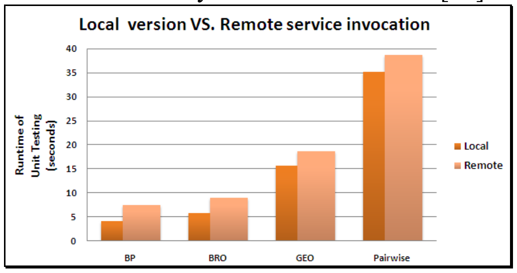

Figure 8: Local-Remote invocations

Figure 8 shows the time comparison to invoke local and remote test services, and for every test approach, the remote invocation costs 2.5 to 3.5 seconds more than local invocation. The increased time come from two portions: time to communicate between tenants and TaaS, and time to wrap test services into containers. The increased time does not change much among different test services due to: (1) The stable local area network, (2) Apache Tuscany as SCA runtime, which is a lightweight implementation.

In addition, it is observed that black-box services cost more time than white-box services. Because black box services need end users to input some constrains for each parameter manually, and use them for test case generation. GEO needs one input for each parameter while Pair-wise needs multiple inputs. That results in Pair-wise testing service takes more time than GEO testing service.

This second experiment consists of two sub-experiments where two white-box techniques, BP and BRO are selected for t test case generation. The first one shows the growth of runtime when the line of code (LOC) of target under test (TUT) increases while the number of conditional branches remains the same as shown in Figure 9. It is examined that the rate △runtime/△Loc is 0.013 for BP, △runtime/△Loc is 0.017 for BRO. The results are expected, as the complexities of the two test techniques are related to the number of condition statements. The increasing number of LOC just increases parsing time which takes in only a small portion of total test processing time.

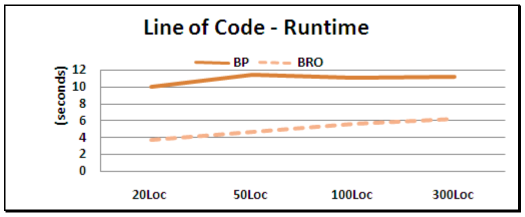

Figure 9: Runtime with increasing of LOCs

The second sub-experiment shows the growth of runtime when the number of conditional branches increases in TUT. Both BP and BRO are all implemented in Genetic algorithm which has random computation in the implementation. We run 10 times for each test case and report the average runtime in Figure 10.

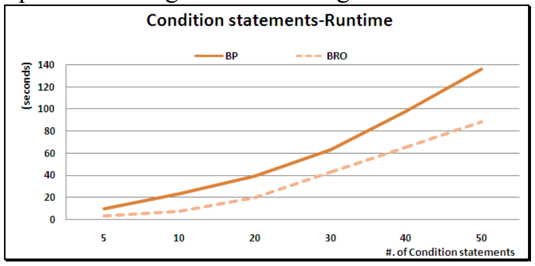

Figure 10: Runtime with increasing of Con. Stm.

It is observed that on average, the rate of △runtime /△condition No. for BP is 2.8, and △runtime/△condition No. for BRO is 1.8. The results are much larger than that of Figure 11, as the two test techniques are sensitive to conditional branches. When the number of conditional branches is below 20, the performance is acceptable for both of them. But for complex code with more 20 conditional statements, it takes longer time to finish testing, which is considered as not good practice.

7. Conclusion and Future Work

This paper presents a framework of a new test model, Testing as a Service (TaaS) to improve the efficiency of software quality assurance. We propose a reference architecture of TaaS, which consists of four layers, GUI layer, Test Service Bus layer, Test Service Composition layer, and Test Service Pool layer. Ontology technique is used to model testing concepts and relations, and SWRL is used to set rules for automatically matching test tasks from tenants with test capability of TaaS platform.

1) Building an ontology model to intelligently match tenant's test tasks with test capability of TaaS platform. OWL classes are used to describe testing domain concepts, OWL properties to express the relationships among the testing concepts and OWL individuals of classes; properties to depict instances of tenant's request; and SWRL to represent the test service matching rules. 2) Allowing community contributors to publish and deploy their test services to TaaS platform. The contributors provide their service meta-data as ontology individuals that are used for matching and reasoning. 3) Automatically compositing test process according to tenant test requests. An abstract test process is defined and automatically configured to achieve tenant's requests based on on-demand reasoning. Three experiments are carried out to show the usage of TaaS platform, compare the performance of local/remote invocations of test services, and examine the trend when increasing the sizes and complexity of target under test. Future work includes incorporating virtual machine techniques to fulfill a variety of test environment requirements of tenants.

Acknowledgement: The work presented in this article is partly sponsored by the National High-Tech Research and Development Plan of China (No. 2006AA01Z175) and IBM 2008 SUR (MOU-PKU-08019).

Reference

[1] Lian Yu, Shuang Su, Jing Zhao, et al, "Performing Unit Testing Based on Testing as a Service (TaaS) Approach", Proceedings of International Conference on Service Science (ICSS) 2008, pp. 127-131.
[2] UDDI: http://uddi.org/pubs/uddi-v3.0.2-20041019.htm
[3] WSRR: http://www-01.ibm.com/software/integration/wsrr/
[4] OWL: http://www.w3.org/TR/owl-features/
[5] SWRL : http://www.w3.org/Submission/SWRL/.
[6] Ontology and Protégé: http://protege.stanford.edu/publications/
[7] Jess: http://herzberg.ca.sandia.gov/jess
[8] Basis Path Testing: http://hissa.nist.gov/basicpathtest/
[9] K. C. Tai, "What to do beyond branch testing", ACM SIGSOFT Software Engineering Notes archive Volume 14 (2), 1989, pp. 58 – 61.
[10] Protégé: http://protege.stanford.edu/.
[11] Jess: http://herzberg.ca.sandia.gov/.
[12] BPEL: http://xml.coverpages.org/WS-BPEL-CS01.pdf
[13] Injection: http://martinfowler.com/articles/injection.html
[14] Tuscany SCA, SDO: http://tuscany.apache.org/
[15] T. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol SE-2, Number 4, December 1976, pg. 308-320.
[16] Abreu, B. T., Martins, E., and de Sousa, F. L. "Generalized Extremal Optimization: a competitive algorithm for test data generation". In proceedings of the 17th IEEE Int. Symposium on Software Reliability Engineering, volume 17, Raleigh, USA, 2006.
[17] A.W. Williams. "Determination of test configurations for pair-wise interaction coverage". In Proceedings of the 13th International Conference on the Testing of Communicating Systems, August 2000, Ottawa, Canada, pp. 59-74.