Collaborative Testing of Web Services

Hong Zhu, *Member*, *IEEE CS*, and Yufeng Zhang

Abstract – Software testers are confronted with great challenges in testing Web Services (WS) because of the lack of software artefacts, control over test executions and means of observation on the internal behaviour of services while they must deal with diversities in the test requirements and service implementation techniques. An automated testing technique must be developed with the capability of testing on-the-fly non-intrusively and non-disruptively. Addressing these problems, this paper proposes a framework of collaborative testing in which test tasks are completed through the collaboration of various test services that are registered, discovered and invoked at runtime using the ontology of software testing STOWS. The composition of test services are realized by using test brokers, which are also test services but specialized in the coordination of other test services. The ontology can be extended and updated through an ontology management service so that it can support a wide open range of test activities, methods, techniques and types of software artefacts. The paper presents a prototype implementation of the framework in semantic WS and demonstrates the feasibility of the framework by running examples of wrapping up a testing tool into a test service, developing a service for test executions of a WS, and composing existing test services for more complicated testing tasks. Experimental evaluation of the framework has also demonstrated its scalability.

Index terms – Software Engineering, Distributed/Internet based software engineering tools and techniques, Testing tools.

1. INTRODUCTION

The research on testing Web Services (WS) has been growing rapidly in recent years [1, 2, 3, 4]. Most research efforts fall into the following classes.

A. Generation of test cases. Techniques have been developed to generate test cases from syntax definitions of WS in WSDL [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17], business process and behavioural models in BPEL [18, 19, 20, 21, 22, 23, 24, 25, 26], ontology based descriptions of semantics in OWL-S [27, 28, 29], and other formal models of WS such as finite state machines and labeled transition systems [30, 31, 32], grammar graphs [33, 34], and first order logic [35], etc. These techniques have addressed various WS specific issues, such as the *robustness* in dealing with invalid inputs and errors in invocation sequences, *fault tolerance* to the failures of other services that it depends on and unreliable communication connections, and *security* in the environment that is vulnerable to malicious attacks, and so on.

B. Generation of testbed. A service often relies on other services to perform its function. However, in service unit testing and also in progressive service integration testing, the service under test needs to be separated from other services that it depends on. Techniques have been developed to generate service stubs [36] or mock services [37] to replace the other services for testing.

C. Checking the correctness of test outputs. Research work has been reported in the literature to check the correctness of service output against formal specifications, such as using metamorphic relations [38], or a voting mechanism to compare the output from multiple equivalent services [39, 40], etc.

- *Prof. Hong Zhu is with the Department of Computing and Electronics, School of Technology, Oxford Brookes University, Oxford OX33 1HX, UK. Email: hzhu@brookes.ac.uk.*
- *Mr. Yufeng Zhang is with the Department of Computer Science, The National University of Defense Technology, Changsha, China. Email: yufengzhang@nudt.edu.cn*

D. Testing tools. A number of prototypes and commercial tools have been developed to support various activities in testing WS, such as Coyote [6], WS-FIT [7], TAXI [41], PLASTIC [42], LTSA-WS [32]; just to mention a few.

However, despite the advances made in the past few years, great challenges remain. In particular, it is still an open question how to cope with the following difficult issues in WS testing [3, 43, 44].

A. The lack of software artefacts. A service oriented application commonly consists of parts (i.e. services) owned by many different stakeholders. Thus, typically, developers of a part have no access to the design document, source code, even the executable code of the other parts. These software artefacts are crucial to perform test activities efficiently and effectively.

B. The lack of control over test executions. A service oriented application is intrinsically distributed, and typically contains parts running on hardware owned by other stakeholders. Thus, a tester often lacks control over the executions of the other owners' parts.

C. The lack of a means of observation of internal behaviour. Another consequence of distributed ownership in service-oriented applications is that testers often lack the means to observe the internal behaviours of the components owned by other ventors.

It is widely recognized that a testing technology for WS must also meet the following requirements.

A. Capability of dealing with diversity. The distributed and shared ownership of services also implies that the parts of a service-oriented application may operate on a variety of hardware and software platforms with different deployment configurations and delivering services of differing quality. Testing has to be performed in a heterogeneous environment. On the other hand, different service requesters may well have different test requirements to meet their own business purposes. Testing must deal with all such varieties and their combinations.

B. Capability of testing on-the-fly. A typical scenario of service-oriented computing is that a service requester

searches for a required function in a registry, and then dynamically links to the service and invokes it. It is widely believed that testing before the invocation is necessary especially in mission critical applications. Such testing, called *testing on-the-fly*, differs from traditional integration testing due to the fact that the time of testing is just before the invocation while all parts to be integrated are already in operation.

C. Capability of performing testing non-intrusively and non-disruptively. A consequence of testing on-the-fly is that, from a service provider's point of view, the test invocations of a service must be distinguished from the real ones so that the normal operation of the service is not interrupted by test activities. On the other hand, from a client's point of view, test invocations should also be distinguished from real ones so that they do not actually receive the real services and do not pay for such test invocations as real services.

D. Capability of full automation. The requirement of testing on-the-fly eliminates the possibility of manual testing. Thus, all test activities must be performed automatically.

It has been recognized that to address all these issues, testing WS should be a collaborative effort contributed to by all stakeholders [40, 41, 44]. In this paper, we present a framework for collaborative testing in which testing activities are accomplished through interactions among multiple participants.

The remainder of the paper is organized as follows. Section 2 outlines the framework and illustrates it with a typical scenario. Section 3 presents a prototype implementation of the framework. Section 4 demonstrates the feasibility of the framework by a run example. Section 5 reports the experiments that evaluate the scalability. Finally, section 6 concludes the paper with a comparison of related works and a discussion of future work.

2. Framework for Testing WS

This section elaborates the framework, illustrates it with a typical scenario and identifies the technical challenges.

2.1. A Typical Scenario

Suppose that a fictitious car insurance broker CIB is developing a web-based system that provides online services of car insurance. In particular, they provide the following services to their end users.

The end users submit car insurance requirements to CIB and get quotes from various insurers that CIB is connected to, and then select one to insure the car. To do so, CIB takes information of the car, its usage, and the payment. It uses the WS of its bank B to check the validity of user's payment information, pass the payment to the selected insurer and takes commissions from the insurer and/or the user. The car insurance broker's software system has a user interface to enable interactive uses, and a WS interface to enable other programs to connect as service requesters. Its binding to the bank's WS is static. However, since insurance is an active business domain, new insurance providers may emerge and existing ones may leave the market from time to time, the broker's software binds dynamically to multiple insurance providers to ensure that the business is competitive on the market. The structure of the system is shown in Fig. 1.

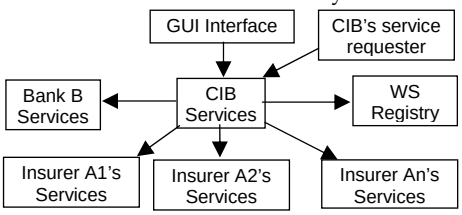

Fig. 1 Structure of Car Insurance Broker Services

The developer of CIB's service must test not only its own code, but also its integration with other WS, i.e. the WS of the insurers and the bank. This paper focuses on the integration with dynamic binding. The following discusses how the challenges can be resolved in the proposed framework.

2.2. The Proposed Framework

The key notion of the framework is *test services* (*T-service* in short), which are services designated to perform various test tasks [44]. A T-service could be provided by the same organization of their normal services or by a third party that is independent of the normal service provider but specialized in testing. For the sake of clarity, we use *functional service* (or *F-service* in short) to denote the normal services in the sequel.

2.2.1. Service Specific T-services

Ideally, each F-service should be accompanied with a special T-service so that test executions of the F-service can be performed by the corresponding T-service. Thus, the normal operation of the original F-service is not disturbed by test requests and the cost of testing are not charged as real invocations of the F-service. The F-service provider can distinguish real requests from the test requests so that no real world effect is caused by test requests. To ensure the testing carried on a T-service faithfully represent the functional services, the following two principles should be observed in the design and implementation of T-services.

(a) A T-service should act in the same way as its functional service as much as possible so that when a test passed by the T-service implies that the F-service is also correct on the test cases.

(b) A T-service should have a 'firewall' so that effects on the real world are stopped and the normal operation of the F-service is not disrupted.

An implication of principle (a) is that the business logic that a service implements may be duplicated by its corresponding T-service in order to test it adequately. On the other hand, an exact copy of the F-service may not achieve the goal of T-service according to principle (b). It is worth noting that in certain special cases the T-service can be absent and all testing are performed on the F-services. For example, if a service contains no internal state and has no effect on the physical real world, the T-service can be a simple duplicate of the F-service, even

be the F-service itself. When the development and maintenance of a T-service is too expensive, or testing the service on-the-fly is unnecessary, the role of T-service can be performed by the F-service.

A T-service that only provides this test execution function can be regarded as a mock service [37]. However, in addition to this, a T-service accompanying an F-service should also provide further support to other test activities. For example, the formal specification of the semantics of the service, the internal design, such as UML diagrams, of the F-service, the configuration of the hardware and software platform, the service policy, even the source code etc., are of particular importance to testers. These kinds of information can be released to trusted T-services subject to preserve the intellectual property rights and privacy, but withheld from the general public.

Moreover, many test activities rely on the information of system internal behaviours, such as the measurement of code coverage, the checking of the internal states of the program during test executions, etc. These can also be provided by the accompanying T-services. Therefore, the T-service accompanying an F-service can be much more than simply a mock service [37].

2.2.2. General purpose testers

Besides service specific T-services that accompany F-services, a test service can also be a general purpose test tool that performs various test activities, such as test planning, test case generation, and test result checking, etc. A general purpose T-service can be specialized in certain testing techniques or methods such as the generation of test cases from WSDL or BPEL using certain WS testing techniques mentioned in Section 1. For the sake of convenience, such general purposes T-services are also called *general testers* in the sequel to distinguish them from service specific T-services.

2.2.3. Test Brokers

One particular type of general purpose T-services that will greatly improve the collaboration between the parties involved in WS testing is *test broker*. As discussed in Section 1, test tasks are usually too complicated to be performed directly by one T-service. A solution to this problem is to introduce test brokers, which compose and coordinate other T-services to carry out test tasks. Typically, there are multiple test brokers; for example, each specializes in one type of testing processes.

As a coordinator, a test broker receives test requests, decomposes the task into subtasks and generates test plans, searches for capable testers for each subtask, invokes testers and returns test results to users. It controls the process of testing. A test broker not only bridges the gap between the users and testers, it can also monitor the dynamic behaviours of T-services and keep a repository of tests performed on each service for future choices of T-services and optimization of test efforts.

2.2.4. Registry and Matchmaker

In our framework, T-services interoperate with each other via SOAP messages. They need to advertise their service descriptions in a service registry to be discovered and invoked at runtime to achieve testing on-the-fly with a high degree of automation. Because of the complexity of the semantics of the service descriptions, we use Semantic WS registry to register T-services, which is composed of a UDDI registry and a Matchmaker [45].

Fig. 2 illustrates the structure of the framework.

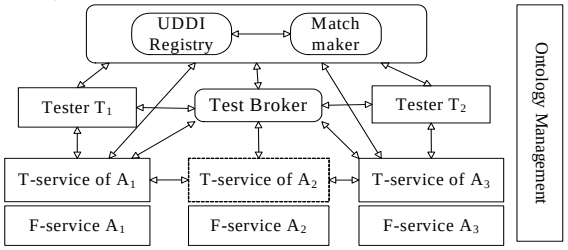

Fig. 2 Reference Architecture of the Framework

2.2.5. Ontology Manager

We use an ontology of software testing to provide a standard set of vocabulary for encoding the information passed between T-services. This makes automatic processing of test tasks feasible.

The extendibility of our framework is achieved by dynamic management of the ontology through another special service, i.e. the ontology management service (OMS). It provides services to update the ontology.

It is worth noting that, first, the framework focuses on the management aspect of testing rather than any specific testing techniques or tools. Most existing works on WS testing are complementary to our framework in the sense that their methods, techniques and tools can be implemented as T-services. The framework facilitates their integration by providing the interfaces and collaboration mechanisms and ensuring the availability of software artefacts that they require. The loosely coupled framework lays a foundation for composing various T-services by the utilization of Semantic WS technology.

Second, the framework is based on the model of WS applications as a network of services interconnected through messages, where services can be dynamically discovered and linked to at run time. The internal structure that a service is implemented is not taken into consideration in the model. This has two implications. First, the framework can be applied to all services that are implemented with any internal structure. Therefore, it is generally applicable. On the other hand, the technology neither takes the advantages of the information about the internal structure of services, nor addresses testing problems due to such internal structure.

2.3. Illustration in the Typical Scenario

We now illustrate how the framework addresses the issues in testing WS using the scenario given in section 2.1.

2.3.1. Architecture

By applying the framework to the scenario, we have the following architecture shown in Fig. 3, where normal service invocations are depicted in solid line arrows and

T-service invocations are denoted by dash line arrows.

In particular, each of the bank *B*'s WS, CIB's WS and insurer A_i's WS has an accompanying T-service. These T-services are registered to the UDDI registry. A test task can be accomplished through collaborations between these T-services.

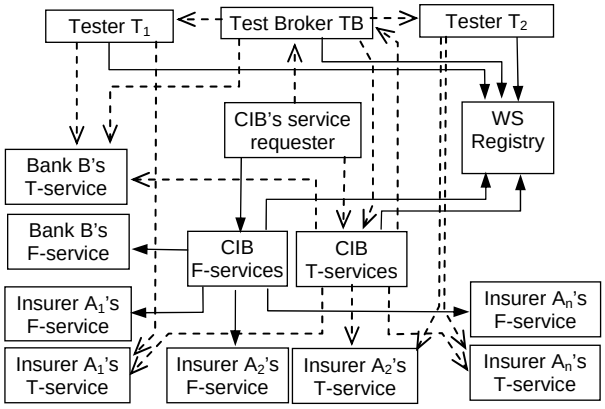

Fig. 3 System Architecture in the Typical Scenario

2.3.2. Collaboration process

Consider the situation that the *CIB* intends to establish a dynamic composition with insurer *A* and to test the service on-the-fly. It delegates the testing task to a test broker *TB*. Fig. 4 shows a typical example of collaboration processes managed by *TB*.

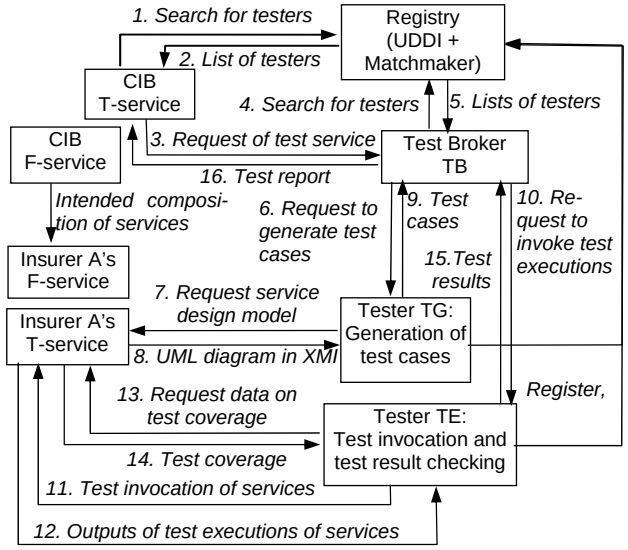

Fig. 4 The Collaboration Process in a Typical Scenario

The process starts with the generation of a test task by *CIB's* WS and a search request for finding a proper tester is submitted to the service registry. The search request message should contain the information about the capability of the required tester. The search result is a list of testers, from which a test broker *TB* capable of handling the task is selected. The test request is then sent to *TB*. The test request message contains the information about the test task including the service to be tested, the test adequacy to achieve, and the criterion for checking output correctness, etc.

The test broker *TB* decomposes the test task into a sequence of subtasks and searches for appropriate testers for each subtask by submitting search requests to the registry. It then selects one tester for each subtask. In this example, we assume two testers *TG* and *TE* are selected. The former performs the sub-task of test case generation and the latter invokes test executions, checks the correctness of test output and measures the test coverage. To generate test cases, *TG* sends a request to insurer *A*'s T-service to obtain its design model. After checking the trustworthy of tester *TG*, the insurer *A*'s T-service releases its design model to *TG*. After successfully obtained the design model, *TG* produces a set of test cases and returns a test suite to the test broker *TB*. The test broker then passes the test cases to *TE*, requests for the test invocation of the insurer *A*'s services using the test cases and requests it to check the output correctness and to measure the test coverage. *TE* performs these tasks by collaboration with the insurer *A*'s T-services. The test results are then returned to the test broker *TB*. Finally, *TB* assembles a test report containing information about test output correctness and test adequacy. The test report is sent to *CIB*, which is used to determine whether the dynamic link will take place.

2.4. Key Technical Issues

From the illustrative scenario given above, we can identify a number of technical issues that are crucial to the practical implementation of the framework.

2.4.1. Semantic complexity of communications

The various parties that participate in the registration, discovery and invocation of T-services communicate with each other through SOAP messages. These messages are complex in semantics. In particular, a T-service must publish its services with a clear and accurate description of its capability so that capability-based search of testers can be performed. The diversity of testing methods, test activities, test environments, and software artefacts used and produced in testing make the description of capability very complicated. Searching for appropriate T-services for a test task must match test tasks with T-service capabilities. This is also a complicated issue since test tasks are not in one-one correspondence to capabilities. Finally, test tasks must be submitted to T-services with parameters of a wide range. Typically, a test task involves multiple software artefacts, such as test cases, the service to be tested, output of test executions, the test oracle to check correctness of output, and so forth. The parameters are therefore often of high complexity.

To deal with semantic complexity, we employ ontology of software testing. In general, ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for the combination and extension of the vocabulary [46, 47]. In Section 3.1, we will demonstrate that the ontology of software testing developed in agent-based approach to software testing [69, 70] can be easily adapted for testing WS and implemented in Semantic WS technology. It provides a set of

standard terminology for T-service registration, discovery and invocation.

2.4.2. Process complexity of interactions

Test processes are complicated, too. Test activities in a testing process are usually interdependent and must be performed in the right order. A variety of testing tools must be used and T-services invoked. Failure to perform a test task may also occur for many different reasons. Thus, interactions among these services may be very complicated as well. Controlling and monitoring such complicated processes thus play a key role to the success of the framework.

Our solution to this issue is to employ a special kind of T-services, called test brokers, to control and monitor the testing process using appropriate mechanisms, such as service orchestration and chorography. Great efforts have been reported in the literature on service composition. We believe that such approaches are applicable to our framework because the framework does not require any changes to the service oriented architecture.

2.4.3. Extendibility and flexibility of the framework

Because of the rapid development of WS technology and expansion of its application areas, the ontology of software testing must be extendible and flexible in order to cope with new testing techniques and methods, new test requirements, and new software artefacts under test and/or their presentation formats.

Our solution is a centralized public service of ontology management. Users of the ontology can extend the ontology by submitting extension request to add new concepts and relationships to the ontology. However, extension, revision and deletion of existing concepts and relations must under tight control so that the registration, discovery and invocation of existing T-services are not affected by changes to the ontology.

3. IMPLEMENTATION IN SEMANTIC WS

The Web Ontology Language OWL is a semantic markup language for publishing and sharing ontologies on the World Wide Web [48]. It is designed for applications that need to process the content of information. It is a part of the growing stack of W3C recommendations related to the Semantic Web. In this section we adapt the ontology of software testing developed in [69, 70] and discuss how it is implemented in OWL.

3.1. STOWS: Ontology of WS Testing

STOWS (Software Testing Ontology for WS) is proposed in [44] based on the ontology developed in [69, 70]. It was adapted for WS testing.

Concepts in STOWS are classified into three categories: elementary concepts, basic testing concepts and compound testing concepts.

The elementary concepts are those general concepts about computer software and hardware based on which testing concepts are defined. They include the simple objects involved in software testing, such as the types of *Hardware* and *Software* artefacts and their *Format*, etc.

The basic testing concepts include *Tester*, *Artefact*, *Activity*, *Context*, *Method*, and *Environment*. These basic concepts are combined together to express compound testing concepts, which include *Task* and *Capability*. The following describes the concepts one by one.

3.1.1. Basic testing concepts

Tester. A tester refers to a particular party who participates in a test activity. Generally speaking, testers can be human beings, organizations and software systems. In the service oriented framework, T-services perform the test tasks, thus they are testers, too. It can be an atomic T-service, or a composition of T-services. One important property of tester is its capability, which reflects the capability to perform test tasks.

Activity. There are various test activities including test planning, test case generation, test execution, result validation, adequacy measurement and test report generation, etc.

Artefact. Various kinds of artefacts may be involved in test activities as input/output, such as test plan, test cases, test results, program, specification and so forth. The most important property of class *Artefact* is *Location,* whose value is an URL referring to the location of the *Artefact*. Each type of artefacts is a subclass of artefact, and inherits the properties from *Artefact*. The subclasses of *Artefact* can be added into the ontology using the ontology management services.

Context. Test activities may occur in different software development stages and have various test *purposes*. The concept context defines the contexts of test activities in testing processes and test methodologies. Typically, the contexts include unit testing, integration testing, system testing, regression testing, etc.

Method. For each test activity, there may be multiple applicable test methods. Method is a part of the capability and also an optional part of test task. Test methods can be classified in a number of different ways. For example, test methods can be classified into program-based, specification-based, usage-based, etc. They can also be classified into structural testing, fault-based testing, error-based testing, etc. Structural testing methods can be further classified into control-flow testing, data-flow testing, etc. Therefore, test methods are represented as a hierarchy in the ontology.

Environment. It is the hardware and software configuration in which a test activity is performed.

3.1.2. Capability of T-services and Test Tasks

The capability of a T-service represents its capability of performing test tasks. The class *Capability* in the ontology defines the aspects that affect the capability of a service to perform tasks, including the activities that the service can do, the test methods that the service uses, the artefacts that the service consumes and produces, the context in which the service performs test activities, and the environment in which test activities are carried out, etc.

Therefore, it is composed from several basic test concepts. The structure of *Capability* is shown in the UML class diagram given in Fig. 5.

Task describes the test task to be carried out. It is used in service invocation. A test task also has six aspects: the activity to be performed, the context of the activity, the required test method and test environment, and the input and output artefacts. The compositions are in the same structure as capability, but have different semantics. The structure is shown in Fig. 5.

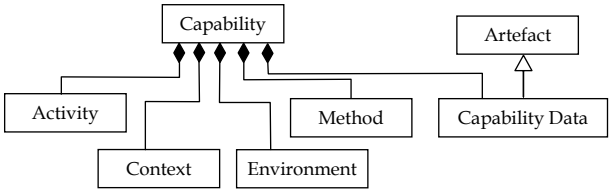

Fig. 5 The Structure of Capability and Task

3.2. Representation of STOWS in OWL

In OWL-S, semantic descriptions are presented in the form of service profiles and used in service registration and discovery. The vocabulary of a subject domain is defined in a data model as classes with subclass relations.

To implement the ontology STOWS, we represent the concepts, including elementary, basic and compound concepts, as classes in OWL data model. To use the ontology for the registration, discovery and invocation of T-services, the compound concepts capability and task are transformed into service profiles. In OWL-S, a service profile contains the IOPR (Inputs, Outputs, Preconditions and Results) and a classification of the service [49]. Fig. 6 shows how the concept of capability is represented in service profile.

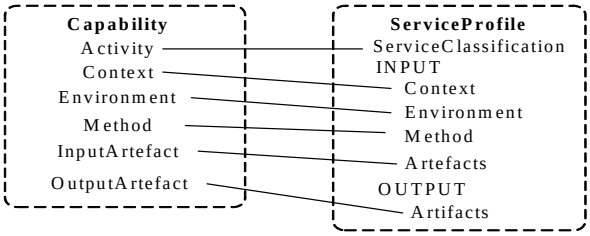

Fig. 6 Mapping Between Capability and Service Profile

```
<profile:Profile rdf:about="#testcase_generation">
    <profile:serviceClassification rdf:datatype=
          "http://www.w3.org/2001/XMLSchema#anyURI">
          http://... /testingontology.owl#TestCaseGeneraton
    </profile:serviceClassification>
    <profile:hasInput>
        <process:Input rdf:ID="input_program">
            <process:parameterType rdf:datatype=
                "http://www.w3.org/2001/XMLSchema#anyURI">
                http://.../testingontology.owl#Program
            </process:parameterType>
        </process:Input> </profile:hasInput>
    <profile:hasOutput>
        <process:Output rdf:ID="output_testcase">
            <process:parameterType rdf:datatype=
                "http://www.w3.org/2001/XMLSchema#anyURI">
                http://.../testingontology.owl#TestCase
            </process:parameterType> </process:Output>
    </profile:hasOutput>
</profile:Profile>
```

Fig. 7 An Example of Service Profile

In the service profile of T-service, the test context, the environment and the method aspects are represented as input parameters *Context*, *Environment* and *Method*. For example, Fig. 7 shows a part of a service profile, whose *serviceClassification* is *TestCaseGeneration*. The *hasInput* and *hasOutput* properties indicate that the service takes a *Program* as input and produces *TestCase* as output. By representing capability and task concepts in profiles, OWL-S/UDDI Matchmaker can be employed to perform semantic-based search of T-services.

3.3. T-Service Registration and Discovery

The OWL-S/UDDI Matchmaker (Matchmaker for short) extends UDDI registry with a capability based service matching engine [45, 50]. It provides three levels of matching between capability and search request.

a) *Exact matching*: the capabilities in the registry and in the request match exactly.
b) *Plug-in matching*: the service provided is more general than that in the request.
c) *Relaxed matching*: there is a similarity between services provided and that in the request.

The Matchmaker also provides filters for users to construct more accurate service discovery: which are namespace filter, domain filter, text filter, I/O type filter and constraint filter [51]. With these filters, users can construct necessary compound filters to control the precision of matching. The matching engine returns a numeric score for each candidate so that the higher the score, the more similar between the candidate and the request. Therefore, selection from the candidates can be based on the scores that tagged by the Matchmaker on the candidate services.

We have used Matchmaker to enhance the registration and discovery of T-services with semantic information. A T-service provider must first register the service with its profile that defines its capability by using the API provided by the Matchmaker. A service search request is also submitted to the Matchmaker.

3.4. Ontology Management Service (OMS)

The terms used in the capability and test task description must be first defined in the ontology. However, it is impossible to build a complete ontology of software testing given the huge volume of software testing knowledge and the rapid development of new testing technique, methods and tools. Therefore, the ontology must be extendable and open to the public for updating. An ontology management mechanism is provided to enable the population of the ontology. It is delivered as a WS to facilitate the public access to the mechanism.

The ontology management service (OMS) is implemented using the Protégé-OWL API, which is an open source Java library for OWL and RDF. Using the API, OWL data model stored in OWL files or databases can be loaded, changed and saved, queries be made, and reasoning performed using a description logic inference engine [52]. Therefore, the manipulation of the ontology can be implemented as operations on OWL files. Fig. 7 shows the structure of OMS.

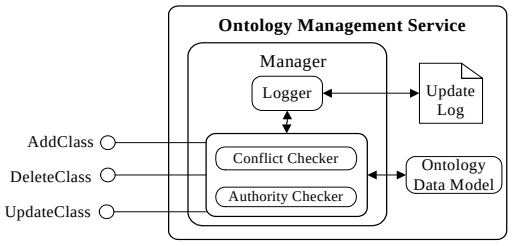

Fig. 8 The Structure of OMS

OMS provides a WS interface to read and update the ontology data model, which is open to the public. The kernel of OMS is the *Manager* module. It provides three services to users: *AddClass*, *DeleteClass* and *UpdateClass* to add new concept, delete concept and revise concept of the ontology.

For example, suppose that a T-service is developed to generate test cases using a new method not included in the ontology, say data mutation. Then, a new test method name '*DataMutation*' can be added to the ontology as a subclass of test method. If a new T-service is to be registered that generates test cases from a new formal specification language called FSL, then a new type of software artefacts called '*FSL*' can be added to the ontology as a subclass of software artefact, rather than a subclass of test method. The relationship between classes in Ontology is represented as properties of classes. Adding or removing a relation can be done by applying operations on the ontology file via OMS. For example, if a *subsumes* relation from branch testing to statement testing is to be added, a '*Subsumes*' property can be added to class '*BranchTesting*' with the value that refers to the class '*StatementTesting*'.

However, to prevent misuses of the ontology management service, restrictions on the manipulation of the data model are imposed through two technical solutions.

First, we classify the classes in the ontology into two types: *elementary classes* and *extended classes*. Elementary classes are those that form the framework of the ontology STOWS. None of them could be pruned down from the ontology hierarchy to avoid structural damage to the ontology. The extended classes are those classes attached to the elementary classes to populate the ontology with concrete concepts and instances of the concepts. They can be added by the users and deleted from the hierarchy. We have implemented an *Authority Checker,* which checks delete operations to ensure that the class to be deleted is *extended class*.

Second, we have also implemented a *Conflict Checker*, which checks the operations on the ontology to ensure that the new class to be added does not exist in the ontology and that the class to be deleted has no subclasses in the hierarchy.

Due to the openness of ontology management, there is a risk of errors caused by update during task executions. If the update is only to add a new concept or relation to the ontology, there should be no effect on existing tasks and services, thus no risk of such errors. However, if the update changes or deletes an existing concept or relation, a task running at the time of update may be affected if messages of the task use the changed concept or relation and rely on the ontology to understand the messages. In such cases, errors may occur due to the updates during execution. How to prevent such errors and reduce the risk of such errors remains an open question that deserves further research.

3.5. Composition of T-Services Using Brokers

As discussed in Section 2.4.2, the interactions among T-services can be complicated. The composition of T-services is a key technical issue for the success of the framework. One of the most promising mechanisms of service collaboration is service brokers. This mechanism is also well supported by OWL-S [53]. However, different from the approach taken in [53], our test brokers do not play the role of registry. A test broker is just a T-service composition and it self is a T-service as well. There may be multiple test brokers owned by different vendors.

We have developed a prototype test broker to demonstrate the feasibility of the approach. Fig. 9 shows the architecture of our prototype test broker. It receives test tasks from service requesters, decomposes a test task into a sequence of subtasks, sets a test plan, searches for other T-services capable of performing the subtasks, and then invokes the T-services according to the plan to carry out the subtasks and passing information between them. Finally, it assembles the results from the services and reports to the service requester. The broker is composed of the following four modules.

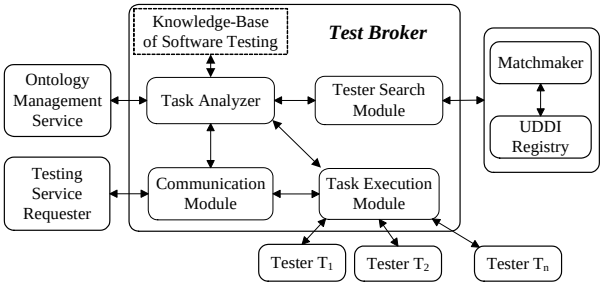

Fig. 9 The Structure of a Test Broker

Communication Module provides an interface to the users. It receives test requests in the form of test tasks and sends out test results also in SOAP format. It transfers test tasks to Task Analyzer and gets test results from the Task Execution Module. Failures to fulfill test requests are also reported to the requesters through this module.

Task Analyzer decomposes a test task into several subtasks and produces test plans according to codified knowledge of software testing processes. It also keeps the track record of test plan executions for each task so that back tracking can be made when a subtask fails.

Tester Search Module searches for testers for each subtask in the test plan generated by the Task Analyzer. A failure to find a suitable tester for a subtask is reported to the Task Analyzer and an alternative test plan may be generated or the whole testing process fails.

Task Execution Module executes the test plan by invoking the testers and passing information between them. A

failure to carry out a subtask is reported to the Task Analyzer and an alternative tester will be employed if any, or an alternative test plan is generated if possible. Otherwise, the whole testing process fails.

The knowledge-base of software testing processes plays a central role in the test plan generation. It can be considered as a finite set of templates of test plans with parameters like task, input and output artefacts. A test task is then checked against the templates one by one and a test plan is produced by instantiating the template when a match is found. Each template can be regarded as a collaboration pattern of T-services. They can also be regarded as heuristic rules about how to compose and coordinate T-services. This significantly reduces the size and complexity of the space in which T-services are searched for and combined. Therefore, the complexity of T-service composition and collaboration can be reduced.

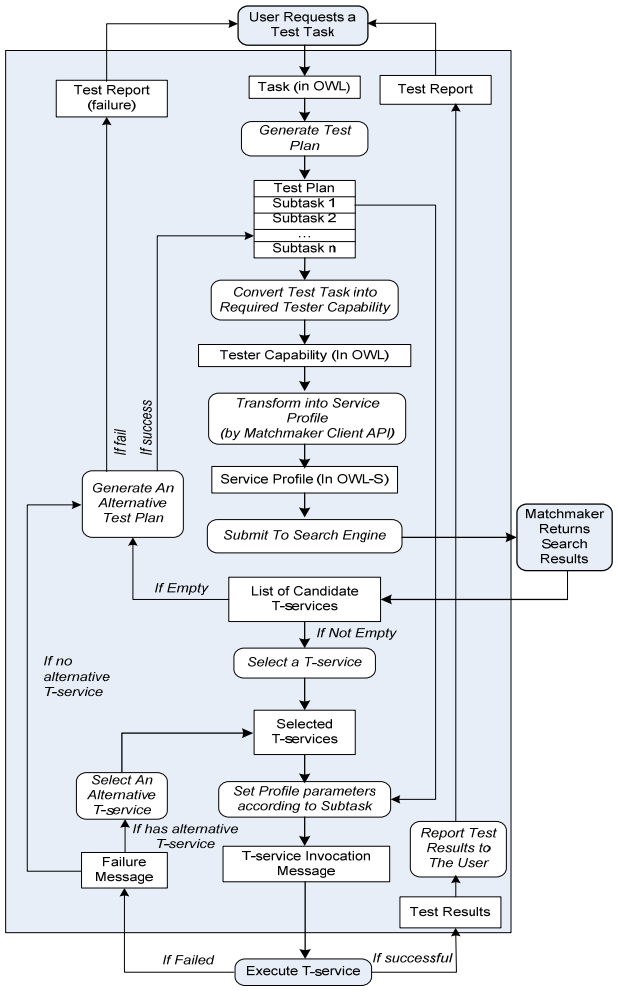

Fig. 10 Process Model of Test Broker

Fig. 10 shows the process that the test broker interacts with Matchmaker and other T-services.

It is worth noting that test tasks and capabilities have the similar structure and the corresponding semantics so that test requests (i.e. test tasks) can be easily transformed into search request (i.e. tester capabilities). Similarly, tester capabilities can be transformed into test subtasks according to the test plan and submitted to the testers. In the implementation of the prototype test broker, we used the Mindswap OWL-S API to parse task and capability profiles and to invoke T-services automatically [54].

4. A Running Example

In this section, we demonstrate the feasibility of the propose approach by a running example.

As shown in Fig. 11, the running example consists of the following WS.

- TCG: a general purpose testing tool that generates test cases from algebraic specifications. It is obtained by wrapping the CASCAT software tool [55,56] into a web service. CASCAT is an automated tool for testing Enterprise Java Beans.
- NCS: a web service that provides numeric calculations of complex numbers. It is the web service to be tested in this example.
- T-NCS: the service specific T-service for NCS. It provides test execution service for testing NCS.
- TFT: a test case format transformer that transforms test cases in the format of CASCAT output into test cases acceptable by T-NCS.

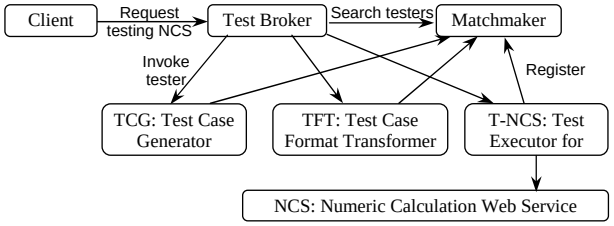

Fig. 11 The Web Services in the Running Example

The following gives some technical details of the registration, search and invocation of the testers in the running example. More details can be found in [71].

4.1. The Registration of Testers

We have built a UDDI registry server using Matchmaker. The web services involved in this running example are all registered on this UDDI registry.

Take the registration of TCG as an example, the service takes a CASOCC specification file as input and generates test cases as output. These artefacts are stored in files and referred to through URLs to the file locations.

To describe this service, the following new classes were added into the ontology.

– *CasoccSpecification*: a subclass of *Specification* that stands for algebraic specification in CASOCC.
– *ComponentTest*: a subclass of *Context* that stands for component testing.
– *CASOCCmethod*: a subclass of *Method* that stands for the method of test case generation from CASOCC.

In its service profile, the *serviceClassification* is set as *TestCaseGeneration*. The Input artefact is specified as the class *CasoccSpecification*. As described in the previous section, the service profile has three parameters that represent the aspects of the service capability. The context of

TCG is *ComponentTest*. Its environment is *notLimited*, which is a subclass of *Thing* (the common ancestor of all the classes in OWL) and represents no requirement on a certain aspect. Its method is *CASOCCmethod*. The output artefact is *TestCase*.

4.2. Submitting Test Tasks

The web service Client plays the role of test requester. It constructs test tasks and submits them to the test broker, which in turn generates requests according to the test tasks and submits them to the Matchmaker to search for T-services. Fig. 12 shows a test task that Client generated and submitted to the test broker requesting test NCS against an algebraic specification written in CASOCC. The input artefact of the task is of type *CasoccSpecification*, and the output artefact type is *TestResult*.

```
<Task rdf:ID="thirdTask">
    <hasContext>
        <ServiceTest rdf:ID="serviceTest"/> </hasContext>
    <hasMethod rdf:resource="# CASOCCBasedMethod "/>
    <hasEnvironment rdf:resource="#notLimited"/>
    <hasActivity rdf:resource="#multiactivites"/>
    <inputArtefact>
        <CasoccSpecification rdf:ID="casoccSpecification">
            <Location rdf:datatype=
            "http://www.w3.org/2001/XMLSchema#anyURI">
                http://.../specification/Calculator.asoc
            </Location> </CasoccSpecification> </inputArtefact>
    <outputArtefact>
        <TestResult rdf:ID="testresult">
            <Location rdf:datatype=
            "http://www.w3.org/2001/XMLSchema#anyURI">
                http://.../artefacts/testresult/fictitioustestresult.txt
            </Location> </TestResult> </outputArtefact>
    <testObject>
        <TestObject rdf:ID="calculateService">
            <operationName rdf:datatype=
            "http://www.w3.org/2001/XMLSchema#string">
                Add </operationName>
            <endpoint rdf:datatype=
            "http://www.w3.org/2001/XMLSchema#string">
                http://.../axis/services/CalculatorImpl
            </endpoint> </TestObject> </testObject>
</Task>
```

Fig. 12 The Task to Test NCS Based on Algebraic Specification

4.3. Search for Testers and Decomposition of Task

Once the test broker receives the test task, it generates a capability description from the test task and constructs a service profile according to the mapping given in Fig. 6. It then calls the Matchmaker to search for T-service providers. In this case, there is no tester that is capable of directly fulfill the test task. Thus, the test broker decomposed it into subtasks and generated a test plan that consisted of three subtasks:

Subtask 1: Generating test cases from the specification. The input artefact of the task is of type *CasoccSpecification*. The output of this subtask is of type *CasoccTestCase*.

Subtask 2: Transforming the test cases into the format that are executable by TesterA. Its input is of type *CasoccTestcase* and output is of type *CalculatorTestCase.*

Subtask 3: Executing test cases and report test results. Its input is of type *CalculatorTestCase* and its output artefact type is *TestResult*.

For each subtask in the test plan, the broker translates it into the corresponding capability description and constructs a service profile. The test broker then submitted the service profile to the Matchmaker to search for appropriate testers. In this case, testers TCG, TFT and T-NCS were discovered for the subtasks, respectively. The test planning finished with each subtask associated with a tester, and the test plan was passed to the execution module for executing the subtasks.

4.4. Invocation

The task execution module of test broker called the testers associated to each subtask according to the order given in the test plan. Data were passed from one subtask to another by the construction of invocation message to the testers. In particular, the output artefact of a subtask was passed to the next subtask. The output of the third subtask was the final result of the test, which was an OWL object. It was returned to the client by the broker.

5. Evaluation

To further evaluate the practical usability of the proposed approach, we have conducted a case study and some experiments to address the following research questions.

a) Can a wide range of software testing tools be integrated into the framework?
b) Can subtle differences between test services be processed adequately?
c) Can the system scale up efficiently?

5.1. Case Study: Dealing with Diversity

To evaluate test brokers' capability of dealing with the diversity of testers, we selected a wide range of software tools reported in the literature to see if they can be turned into testers. Table 1 gives the tools in the case study together with 3 testers mentioned in Section 4.

In the case study, we have successfully described their functionalities in the STOWS ontology and registered to the Matchmaker. For those tools we can get the source code or executable code, they were also wrapped into web services and deployed to a server. For the others, stubs were deployed, which are marked with an asterisk in Table 1.

Table 1. Testers integrated in the framework

Name	Description
CASCAT [56]	A CASOCC-based test case generation tool
Test Case Format Translator	Translate the test case generated by CASCAT into the format recognizable by Calculator Test Case Executor
Test Case Executor	Executes test case for a numeric calculator web service
Klee [57]	Generate and execute test cases from C source code by symbolic execution
Magic [58]	Check conformance between component specifications and their implementations
XML Comparator	Compare XML files
Java NCSS [59]	Measure two standard metrics for Java program
Findbugs [60]	Find bugs in Java program by static analysis
PMD [61]	A static analysis tool for finding potential bugs and other problems in Java source code
WSDL Based Test Case Generator* [5]	A WSDL based test case generation tool
Web Service Test Case Executor* [5]	Execute the test case generated by WSDL Based Test Case Generator

Moreover, for each of these T-services, we have successfully conducted experiments with the broker to search and invoke them in both simple and complex tasks. For example, Klee was selected and invoked by the broker to execute tasks that require C source code as input, test result as output and *symbolic execution* as method.

This case study shows that the approach is capable of dealing with a wide range of software testing, verification and validation tools.

5.2. Experiment 1: Dealing with Subtle Differences

Experiments have also been conducted to test the system's capability of dealing with subtle differences between testers so that appropriate ones are selected accurately.

In this experiment, we applied the data mutation testing technique [62] to generate a large number of service profiles (called mutants) from a set of original profiles (called the seeds). These mutants differ from the seeds on only one parameter. They were generated by applying data mutation operators to a seed on the parameter.

Let x be one of S (for service classification), I (for input artifact, O (for output artifact), M (for method), C (for context) and E (for environment). The data mutation operators are defined as follows.

- *RxF*: Replace the x parameter in the seed profile (a class in the ontology) by its father in the ontology;
- *RxS*: Replace the x parameter in the seed by one of its subclasses in the ontology;
- *RxB*: Replace the x parameter in the seed by one of its brother classes in the ontology;
- *RxN*: Replace the x parameter in the seed by a class in the ontology that has no relation to the parameter.

For instance, if the *ServiceClassification* of a seed profile is *TestCaseGeneration*, then the *RSB* operator will generate profiles with *TestCaseExecution* and *TestResultValidation* as *ServiceClassification*, respectively, assumed that both classes are the sibling classes of *TestCaseGeneration* in the ontology. Sometimes a data mutation operator is not applicable, because, for example, it may have no alternatives in the ontology.

We have used the profiles of the testers in our previous case studies as the seeds and generated 167 mutants in total. These mutants were then registered to the Matchmaker.

An experiment was then conducted to search for testers with search requests matching the seeds and to see if mutants could be filtered out.

The results of the experiment shows that the intended original profiles were always selected. The mutants were never chosen to execute test tasks although sometimes they are included in the search results together with their seeds. In particular, the search scores of these mutants turned out to be as follows:

- Mutants generated by *RxS* operators (i.e. *RSS*, *RIS*, *ROS*, *RMS*, *RCS*, *RES*) have scores that are 1 point less than that of their seeds;
- Mutants generated by *RxF* operators have scores that are 2 points less than that of their seeds.
- Mutants generated by *RxB* and *RxN* (except *ROB* and *RON*) have scores that are 3 points less than that of their seeds.
- Mutants generated by *ROB* and *RON* operators were given much lower scores than their seeds by the Matchmaker.

This experiment shows that test services of subtle differences can be dealt with in our framework satisfactorily.

5.3. Experiment 2: Scalability

Experiments have also been conducted to evaluate the scalability of test brokers in terms of its efficiency to deal with test problems of practical sizes.

We identified three key dimensions of the sizes of testing problems that may affect usability:

- *The number of testers* registered in the registry affects the time needed to search for a tester.
- *The size of the knowledge-base* of software testing affects the time needed to analyze a test task and to decompose it into a sequence of subtasks. In the experiment, we measure the size in terms of the number of test plan templates in the knowledge-base.
- *The complexity of test task* also affects the time needed to process the task. In the experiment, we use the number of different types of subtasks that it will be decomposed into as a simple measurement of its complexity.

The experiments aim at estimating how broker's execution time depends on these factors. For each factor, we designed a set of contexts that each consists of a set of testers, a set of templates and a task. In each context, the broker was run for a number of times. The lengths of time spending on executing various modules of the test broker in the context were collected and their averages were calculated to reduce the random effect of the underlying system software and the network connection. The contexts were constructed using the following repository.

- *Domain of testers*: A repository of testers used in the experiments contains the testers developed in experiment 1, i.e. the set of real testers used in case studies plus their mutants. It contains a total of 178 testers.
- *Domain of test plan templates*: A repository of test plan templates was also generated by applying the data mutation technique on the real templates of our test broker. A large number of mutants were generated. 500 templates in total were actually used in the experiments.
- *Domain of test tasks*: A small repository of test tasks were manually generated so that each task can be decomposed into n different types of subtasks (n=1,2,…,5) according to the real test plan templates.

The designs of the experiments and their main results are as follows.

(1) The effect of the number of testers

To find out how the number of testers registered in registry affects the execution time of the test broker, we selected at random subsets of mutant testers in the repository plus 11 real testers given in Table 1. The sizes of these subsets increase from 20 to 178 in a step of 10. In each case, the set of testers were registered to form a registration state. A set of 11 test tasks were constructed so that each task can be fulfilled by one real tester.

In each registration state, every test task was submitted to the broker to search for the real tester that matches the task. The task is executed repeatedly for 30 times and the average lengths of execution time were calculated to reduce random effects. Experimental data given in Fig. 13 shows that the average search time over 30×11 executions increases with the number of testers in the registry, but in almost a linear manner.

Note that, our experiment results also show that the search time is independent of the test tasks. The details are omitted for the sake of space.

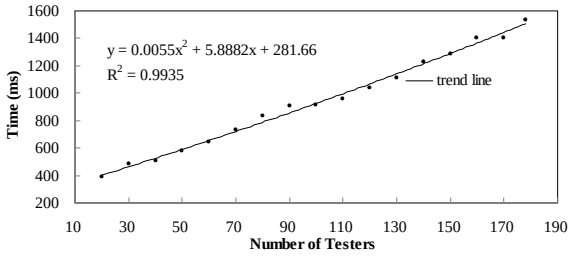

Fig. 13 Time spent on searching for testers

(2) The Effect of the number of test plan templates

The number of the templates in the knowledge base only affects the time spent by the task analyzer module of the broker. In the experiment, we selected at random sets of mutant templates plus one real template that matches the test task as the knowledge base. In each case, we place the real template at the end of the mutants to ensure that the longest time that the task analyzer will execute using the knowledge base.

As shown in Fig. 14, when the size of knowledge base increases from 20 to 500 the time spent by the task analyzer module also increases, but in an almost linear rate.

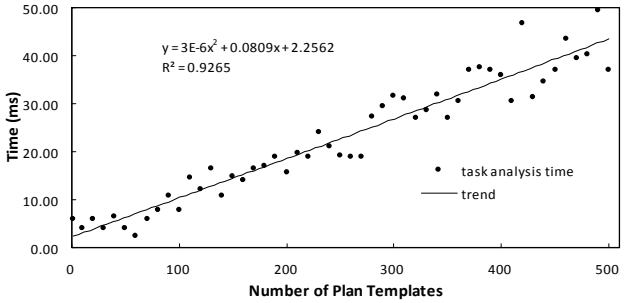

Fig. 14 Time spent by task analyzer

(3) The effect of task complexity

In this experiment, we generated a set of test tasks that are decomposed into different number of subtasks, which ranges from 1 to 5. The test broker was run on each task for 30 times. The lengths of execution time were collected and their averages were calculated.

Note that in comparison with simple tasks that can be fulfilled directly by a tester, the test broker spends more time in processing a complex task on:

a) the generation of task plan ,

b) the searching for testers of subtasks,

c) the execution of subtasks, which can be further split into three parts:

i) the broker prepares data for invoking testers,

ii) the testers fulfill the subtasks, and

iii) the broker receives and unpacks the returned data from the tester.

The time spent on c.ii) only depends on the performance of the tester(s). It is irrelevant to the efficiency of the broker. Therefore, it is omitted in our experiment.

Fig. 15 shows the average lengths of execution times on different tasks with the number of different types of subtasks ranging from 1 to 5. A quadratic polynomial figure fits the curve very well with R^2=0.9984.

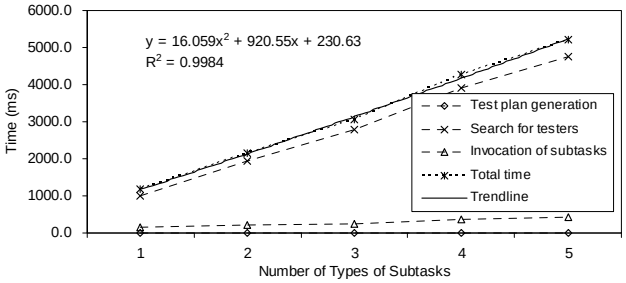

Fig. 15 Time dependence on the number of subtasks

In summary, the experiments show that the broker is capable of dealing with test problems of practical sizes with respect to the number of testers registered, the size of the knowledge-base, and the complexity of test tasks.

6. CONCLUSION

In this paper, we presented service oriented architecture for testing WS. In this architecture, various T-services collaborate with each other to complete test tasks. We employ the ontology of software testing STOWS to describe the capabilities of T-services and test tasks for the registration, discovery and invocation of T-services. The knowledge intensive composition of T-services is realized by the development and employment of test brokers, which are also T-services. We implemented the architecture in Semantic WS technology. Case studies have demonstrated the feasibility of the architecture and illustrated how to wrap up general purpose testing tools and turn them into T-services and how to develop service specific T-services to support the testing of a WS. Experimental evaluation also shows the scalability of the approach.

6.1. Related Work

There are two other frameworks for collaborations in testing WS proposed in the literature.

Since 2003, Tsai and his colleagues and students have advanced a framework of collaboration for WS testing. In 2003 [63], a framework was first proposed, which consists of three types of elements: (a) *test masters* generate test cases based on test scenarios; (b) *test agents* invoke services using test cases generated by test masters; (c) *monitors* catch the data passed between WS and reports state changes to test agents. Tsai *et al*. soon realized that existing UDDI is insufficient to support collaborative testing in their framework and proposed an extension to the function of UDDI to enable collaboration [40, 64]. They proposed to add check-in and check-out services to UDDI

servers so that a service is added to UDDI registry only if it passes a check-in test. A check-out testing is performed every time the service is searched for. A service is recommended to a client only if it passes the check-out test. To facilitate such tests, they require test scripts being included in the information registered for the WS on UDDI. In [39, 40], Tsai and his colleagues went further to investigate the problem how to select a service from a large number of candidates by testing. They developed a test case ranking technique to improve the efficiency of group testing. More recently, in 2007 [65] and 2008 [66], the notion of test broker mentioned in [63] was further developed and a prototype implementation was reported. Test brokers in their framework are much more complicated than ours and aim to achieve many more functions, which include generating test cases, submitting test cases to the WS, coordinating tester and functional services, registering testers and recording the performance of testers, monitoring services and keeping a repository of tests performed for the evaluation of services and testers, etc. A fundamental difference between Tsai *et al*.'s framework and our framework is that they do not use ontology as the definition of the semantics of messages. Search for test services relies on keyword matching. Thus, the framework is weak in dealing with semantic complexity of software testing tasks and its components are more complex without employing semantic WS technology. Another difference is that they do not use knowledge of software testing process to generate test plans. Moreover, their components are not services. In particular, they invoke the real services for testing rather than employ service specific test services.

An approach similar to Tsai et al.'s is taken by Bertolino *et al*. in their audition framework proposed in 2005 [30]. It requires an admission testing when a WS is registered to UDDI. This is equivalent to Tsai *et al*.'s check-in test. But, after a service is registered in a UDDI server, Bertolino *et al*. emphasize on run time monitoring services on both functional and non-functional behaviours, while Tsai *et al.* require check-out tests.

Based on the audition framework, Bertolino and Polini recently proposed a framework called *service test governance* (STG) to incorporate testing into a wider context of quality assurance of WS [67]. Here, governance means imposing a set of policies, procedures, documented standards on WS development, etc. These rules are to be enforced by a certain organization. In addition to the admission test and runtime monitoring, STG also requires WS testing following standard processes.

Although both Tsai *et al*. and Bertolino *et al*. recognised the need of collaboration in testing WS, the technical details about how to collaborate multiple parties in WS testing have not been addressed adequately. Moreover, both frameworks require revisions and extensions of WS standards, especially UDDI. Therefore, as Bertolino and Polini admitted in [67], "*on a pure SOA based scenario the framework is not applicable*".

Even if UDDI can support these frameworks, the extra burden on UDDI servers for performing audition test or check-in/check-out tests will make SOA impractical even if such tasks are delegated to a third party. Further more, it is hard to see how a universal standard on WS testing and quality assurance can fit all purposes and be acceptable by all sectors of industries.

In [68], Tsai *et al*. discussed what is necessary to extend WSDL in order to support testing. They proposed to include the information like input/output dependences between operations, invocation sequences, and structural and functional features of WS. We believe that, although such information is necessary, it is still insufficient to support all aspects of WS testing. On the other hand, it is unnecessary to extend WSDL to provide such information. Information required for testing WS can be provided by using ontology and through T-services.

The framework presented in this paper had its inception in 2006 [44] based on the author's previous work on agent-based approach to testing web-based systems [69, 70]. A preliminary implementation and case study of the framework was reported in [71] without details about the test broker and ontology management service. Our approach differs from the others in that it implements collaborative testing of WS within the framework of service oriented architecture using ontology and also the concept of T-services. In this framework, various testing functions are provided by T-services, such as generating test plan and test cases, invoking test executions, collecting test results, checking output correctness, measuring test adequacy and coverage, and so forth. The collaborations between them are autonomous rather than enforced. That is, what to test and how to test is the choice of the service requester, and what and how to fulfil a client's test request is the choice of test service provider. A T-service requester need to search for T-services, negotiate the cost of test, select a T-service provider and invoke the T-service at runtime. The test activities are then performed by a T-service provider.

This framework is further enriched in this paper by incorporating two facilities. The first is an ontology management facility so that the software testing ontology can be extended and maintained through public services. The second is test brokers, which are also T-services but specialised in the composition of T-services so that complicated testing processes and interactions between T-services can be handled by such professionally developed T-services to simplify the uses of T-services.

The approach has the following advantages in comparison with the existing work. First, it is scalable since T-services are distributed and there is no extra-burden on UDDI servers. Second, this approach can be implemented without any change to the existing standards of Semantic WS [72] as shown in this paper. Third, the need of dealing with variety is achieved through collaborations among many T-services and the employment of ontology of software testing to integrate multiple testing tools. Fourth, the automation of test processes for testing on-the-fly, especially the dynamic composition of

T-services, can be also achieved by employing ontology of software testing and test brokers. Moreover, test executions can be performed by running a separate T-service, thus they do not interfere with the normal operations of the services under test. Finally, when test tasks are performed by a trusted third party of professional T-services, documents and source code as well as other software artefacts can be released to the T-service provider with proper intellectual property protection.

6.2. Future work

The test broker in the prototype is still very primitive. Further research on the design and implementation of more powerful test brokers will have a significant impact on the usability of the T-services. In particular, using knowledge of software testing processes to generate test plans seems a promising topic for further work. Such knowledge can be encoded in a process definition language such as BPEL. Therefore, a much more flexible and powerful test broker can be devised. Another direction to enhance the functionality of test brokers is to associate monitoring functions to brokers as in Tsai *et al*.'s approach so that the previous performance of T-services can be taken into consideration in the selection of testers.

An issue that has not been addressed adequately is the testing of long running processes. A simple solution could be to allow testers to distinguish long running processes from short running tasks either in the test request message (i.e. in the test task description) or in the service description (i.e. in WSDL). An upper limit to the waiting time for test results should then be set accordingly to avoid infinite waiting. The broker could also set different running modes for short and long running tasks. For the latter, the broker may generate a new thread to execute the function. The ability of broker to handle long running processes is related to the platform on which the broker is deployed. The soap engine we used in case study, i.e. Apache Axis 1.4, is capable of supporting this.

Moreover, as discussed in Section 1, a particular difficulty in testing WS is due to the lack of software artefacts to support test activities. The framework presented in this paper offers the opportunity to incorporate a trust mechanism so that design documents, source code and many other types of internal information of services can be delivered to trustable T-services. Further research on how such a trust mechanism to interoperate with the T-services needs to be worked out in detail.

Another hard problem to be solved is associated to the management of ontology. Due to its openness, errors due to update during the execution of a task may occur as discussed in Section 3.4. How to prevent such errors and to reduce the risk is still an open question.

Testing is one of the quality assurance activities for the development of services. It is worth investigating into how to extend and/or adapt the framework for a wider range of quality assurance activities such as static analysis and verification and dynamic monitoring of services, etc. This may need to extend the network model of WS to incorporate the internal structure of services.

Finally, we have only reported the main results of the evaluation case studies and experiments. More detailed will be reported separately due to the limitation of space. Further empirical study and evaluation of the proposed approach should be conducted, especially with regard to the costs associated to the design, implementation and operation of service specific T-services.

REFERENCES

[1] F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, *Web Services Architecture*. W3C Working Group Note, http://www.w3.org/TR/ws-arch, 2004.

[2] M. Stal, "Web Services: Beyond Component-Based Computing", *C.ACM*, vol. 45, no. 10, pp.71-76, Oct. 2002.

[3] G. Canfora and M. Penta, "Service-Oriented Architectures Testing: A Survey," *Software Engineering: Int'l Summer Schools* (*ISSSE* 2006-2008), *Revised Tutorial Lectures*, A. Lucia and F. Ferrucci (Eds.), LNCS vol.5413, Springer-Verlag, pp.78-105, 2009.

[4] M.Bozkurt, M. Harman and Y. Hassoun, *Testing Web Services: A Survey*. Technical Report TR-10-01, Department of Computer Science, King's College London. January, 2010.

[5] X. Bai, W. Dong, W. Tsai and Y. Chen, "WSDL-Based Automatic Test Case Generation for Web Services Testing," *Proc. Of SOSE'05*, pp.215-220, Oct. 2005.

[6] W. Tsai, R. Paul, W. Song and Z. Cao, "Coyote: An XML-Based Framework for Web Services Testing," *Proc. of HASE'02*, pp.173-174, Oct. 2002.

[7] N. Looker, M. Munro and J. Xu, "WS-FIT: A Tool for Dependability Analysis of Web Services," *Proc. of COMPSAC'04*, pp.120-123, Sept. 2004.

[8] J. Offutt and W. Xu, "Generating test cases for web services using data perturbation," *SIGSOFT Softw. Eng. Notes,* vol. 29, no. 5, pp.1-10, Sept. 2004.

[9] S. C. Lee and J. Offutt, "Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis," *Proc. of ISSRE'01*, pp.200-209, Nov. 2001.

[10] W. Xu, J. Offutt and J. Luo, "Testing Web Services by XML Perturbation," *Proc. of ISSRE'05*, pp.257-266, Nov. 2005.

[11] M. P. Emer, S. R. Vergilio and M. Jino, "A Testing Approach for XML Schemas," *Proc. of COMPSAC'05*, pp57-62, Jul. 2005.

[12] A. Bertolino, J. Gao and E. Marchetti, "XML every-flavor testing", *Proc. of WEBIST'06*, INSTICC Press, pp.268-273, Apr. 2006.

[13] A. Bertolino, J. Gao, E. Marchetti and A. Polini, "Systematic generation of XML instances to test complex software applications," *Rapid Integration of Software Engineering Techniques, Third Int'l Workshop, RISE 2006, Revised Selected Papers, Guelfi, N. and Buchs, D. (Eds.),* LNCS vol. 4401, Springer, pp.114-129, 2007.

[14] A. Bertolino, J. Gao, E. Marchetti and A. Polini, "Automatic Test Data Generation for XML Schema-based Partition Testing," *Proc. of AST'07* , p.4, May 2007.

[15] J. B. Li and J. Miller, "Testing the Semantics of W3C XML Schema," *Proc. of COMPSAC'05*, pp.443-448, Jul. 2005.

[16] L. F. de Almeida and S. R. Vergilio, "Exploring Perturbation Based Testing for Web Services," *Proc. of ICWS'06*, pp.717-726, Sept. 2006.

[17] S. Hanna and M. Munro, "An Approach for WSDL-Based Automated Robustness Testing of Web Services," *Information Systems Development: Challenges in Practice, Theory, and Education*, vol. 2, Barry, C. *et al*. (eds.), Springer, pp.493-504, 2009.

[18] J. Garc´ıa-Fanjul, J. Tuya and C. de la Riva, "Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN," *Proc. WS-MaTe*, 2006.

[19] C. Bartolini, A. Bertolino, E. Marchetti and I. Parissis, "Data Flow-Based Validation of Web Services Compositions: Perspectives and Examples," *Architecting Dependable Systems V*, R. Lemos, *et al*. (Eds.), LNCS vol. 5135, Springer-Verlag, pp.298-325, 2008,

[20] C. Bartolini, A. Bertolino and E. Marchetti, "Introducing service-oriented coverage testing," *Proc. of ASE'08*, pp.57-64, 2008.

[21] K. Kaschner and N. Lohmann, "Automatic test case generation for services," *Fourth Int'l Workshop on Engineering Ser-*

vice-Oriented Applications: Analysis and Design (WESOA 2008), Proceedings, LNCS, Springer-Verlag, Dec. 2008.
[22] M. Lallali, F. Zaidi, A. Cavalli and I. Hwang, "Automatic Timed Test Case Generation for Web Services Composition," *Proc. of ECOWS 2008*, pp.53-62, 2008,
[23] Z. Li, W. Sun, Z. B. Jiang and X. Zhang, "BPEL4WS Unit Testing: Framework and Implementation," *Proc. of ICWS'05*, pp.103-110, July 2005.
[24] P. Mayer, *Design and Implementation of a Framework for Testing BPEL Compositions*. Master thesis, Leibnitz University, Germany, 2006.
[25] L. Mei, W. K. Chan and T. H. Tse, "Data flow testing of service-oriented workflow applications," *Proc. of ICSE'08*, pp.371-380, 2008.
[26] Y. Zheng, J. Zhou and P. Krause, "An Automatic Test Case Generation Framework for Web Services," *Journal of Software*, vol. 2, no.3, pp.64–77, 2007.
[27] H. Huang, W. Tsai, R. Paul and Y. Chen, "Automated Model Checking and Testing for Composite Web Services," *Proc. of ISORC'05*, pp.300-307, May 2005.
[28] Y.Wang, X. Bai, J. Li and R. Huang, "Ontology-Based Test Case Generation for Testing Web Services," *Proc. of ISADS'07*, pp.43-50, 2007.
[29] X. Bai, S. Lee, W. T. Tsai and Y. Chen, "Ontology-Based Test Modeling and Partition Testing of Web Services," *Proc. of ICWS'08*, pp. 465-472, 2008.
[30] A. Bertolino and A. Polini, "The Audition Framework for Testing Web Services Interoperability," *Proc. of EUROMICRO'*05, pp.134-142, Aug. 2005.
[31] F. Belli and M. Linschulte, "Event-Driven Modeling and Testing of Web Services," *Proc. of COMPSAC'08*, pp.1168-1173, 2008.
[32] J.Magee, J. Kramer, S. Uchitel and H. Foster, "LTSA-WS: a tool for model-based verification of web service compositions and choreography," *Proc. of ICSE'06*, pp.771-774, 2006.
[33] R. Heckel and M. Lohmann, "Towards Contract-based Testing of Web Services," *Electronic Notes in Theoretical Computer Science,* vol. 82, no.6, 2004.
[34] R. Heckel and L. Mariani, "Automatic conformance testing of web services," Proc. FASE'05, Springer, pp.34-48, 2005.
[35] W.Tsai, X.Wei, Y. Chen, R. Paul, and X. Bai. "Swiss Cheese Test Case Generation for Web Services Testing," *IEICE - Trans. Inf. Syst.,* Vol. 88, no.12, pp.2691-2698, Dec. 2005.
[36] A. Bertolino, G. De Angelis, L. Frantzen and A. Polini, "Model-Based Generation of Testbeds for Web Services," *Proc. TestCom/FATES'08*, pp.266-282, 2008.
[37] H. Huang, H. Liu, Z. Li and J. Zhu, "Surrogate: A Simulation Apparatus for Continuous Integration Testing in Service Oriented Architecture," *Proc. of SCC'08*, vol. 2, pp.223-230, 2008.
[38] W. K. Chan, S. C. Cheung and K. R. P. H. Leung, "A Metamorphic testing Approach for Online testing of service-Oriented software Applications," *Int'l Journal of Web Services Research,* vol. 4, no.2, pp.61- 81, Apr. 2007.
[39] W. Tsai, X. Zhou, Y. Chen and X. Bai, "On Testing and Evaluating Service-Oriented Software," *Computer,* vol. 41, no.8, pp.40-46, Aug. 2008.
[40] W. Tsai, Y. Chen, R. Paul, N. Liao and H. Huang, "Cooperative and Group Testing in Verification of Dynamic Composite Web Services," *Proc. of COMPSAC'04*, vol. 2: Workshops and Fast Abstracts, pp.170-173, Sept. 2004.
[41] A.Bertolino, J. Gao, E. Marchetti and A.Polini, "TAXI--A Tool for XML-Based Testing," *Proc. of ICSE'07* (Companion), pp53-54, May 2007.
[42] A. Bertolino, G. De Angelis, L. Frantzen and A. Polini, "The PLASTIC Framework and Tools for Testing Service-Oriented Applications," *Software Engineering: Int'l Summer Schools, (ISSSE'08)*, pp.106-139, 2008.
[43] G. Canfora and M. Penta, "Testing Services and Service-Centric Systems: Challenges and Opportunities," *IT Professional,* vol. 8, no. 2, pp.10-17, 2006.
[44] H. Zhu, "A Framework for Service-Oriented Testing of Web Services," *Proc. of COMPSAC'06*, pp.679-691, Sept. 2006.
[45] K. Sycara, M. Paolucci, A.Ankolekar and N. Srinivasan, "Automated Discovery, Interaction and Composition of Semantic Web services," *J. Web Semantics,* vol. 1, no.1, pp27-46, Dec. 2003.
[46] M. Uschold and M. Gruninger, "Ontologies: Principles, Methods, and Applications," *Knowledge Engineering Review,* vol.11, no. 2, pp.93-155, 1996.
[47] T.R. Gruber, "A Translation Approach to Portable Ontology Specification," *Knowledge Acquisition,* vol. 5, pp.199-220, 1993.
[48] *OWL Web Ontology Language Reference*. http: //www.w3.org/ TR/ 2004/ REC-owl-ref-20040210/, 2004.
[49] *OWL-S: Semantic Markup for Web Services.* http: //www.w3.org/ Submission/OWL-S/, 2004.
[50] N. Srinivasan, M. Paolucci and K. Sycara, "Adding OWL-S to UDDI, implementation and throughput.," *Proc. The 1st Int'l Workshop on Semantic Web Services and Web Process Composition*, pp.169-182, 2004.
[51] T. Kawamura, J-A. De Blasio, T. Hasegawa, M. Paolucci, and K. Sycara, "A Preliminary Report of a Public Experiment of a Semantic Service Matchmaker combined with a UDDI Business Registry," *Proc. of ICSOC'03*, pp.208-224, Dec. 2003.
[52] *Protégé-owl API Programmer's Guide*. http://protege.stanford.edu/plugins/owl/api/guide.html.
[53] K. Sycara, M. Paolucci, J. Soudry and N. Srinivasan, "Dynamic Discovery and Coordination of Agent-Based Semantic Web Services," *IEEE Internet Computing,* vol. 8, no.3, pp. 66-73, May/June 2004.
[54] *Mindswap OWL-S API*. http://www.mindswap.org/2004/owl-s/api/.
[55] L. Kong, H. Zhu and B. Zhou, "Automated Testing EJB Components Based on Algebraic Specifications," *Proc. of COMPSAC'07*, vol. 2, pp.717-722, 2007.
[56] B. Yu, L. Kong, Y. Zhang and H. Zhu, "Testing Java Components Based on Algebraic Specifications," *Proc. of ICST'08*, pp.190-199, April 2008.
[57] C. Cadar, D. Dunbar and D. Engler, "Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *OSDI*, 2008.
[58] S. C. Edmund, E. Clarke, A. Groce, S. Jha and T. Vienna, "Modular Verification of Software Components in C ," *IEEE Trans. Softw. Eng.* , vol.30 , pp. 388-402, 2004.
[59] JavaNCSS. http://javancss.codehaus.org/
[60] D. Hovemeyer and W. Pugh, "Finding more null pointer bugs, but not too many ," in *Proc. of PASTE'07*, pp.9-14, 2007.
[61] PMD. http://pmd.sourceforge.net/
[62] L. Shan and H. Zhu, "Generating Structurally Complex Test Cases By Data Mutation", *The Computer Journal*, vol.52, pp. 571-588, 2009.
[63] W. Tsai, R. Paul, L. Yu, A. Saimi, Z. Cao, "Scenario-Based Web Services Testing with Distributed Agents," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 10, pp.2130-2144, October, 2003.
[64] W. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi and X. Bai, "Verification of Web Services Using an Enhanced UDDI Server," *Proc. of WORDS'*03, pp.131-138, 2003.
[65] X. Bai, Y. Wang, G. Dai, W.T. Tsai and Y. Chen, "A Framework of Contract-Based Collaborative Verification and Validation of Web Services," *Proc. of CBSE'07*, LNCS vol. 4608, pp.256-271, July 2007.
[66] X. Bai, S. Lee, R. Liu, W.T. Tsai and Y. Chen, "Collaborative Web Services Monitoring with Active Service Broker," *Proc. of COMPSAC'08*, pp.84-91, 2008.
[67] A. Bertolino and A. Polini, "SOA Test Governance: Enabling Service Integration Testing across Organization and Technology Borders," *Proc. of Webtest'09 at ICST'09*, April 2009.
[68] W. Tsai, R. Paul, Y. Wang, C. Fan and D.Wang, "Extending WSDL to Facilitate Web Services Testing," *Proc. of HASE'02*, pp.171-172, 2002.
[69] H. Zhu and Q. Huo, "Developing A Software Testing Ontology in UML for A Software Growth Environment of Web-Based Applications," *Software Evolution with UML and XML*, H. Yang, (ed.), IDEA Group Inc. pp263-295, 2005.
[70] H. Zhu, Q. Huo, and S. Greenwood, "A Multi-Agent Software Environment for Testing Web-based Applications," *Proc. COMPSAC'03*, pp.210-215, Nov. 2003.
[71] Y. Zhang and H. Zhu, "Ontology for Service Oriented Testing of Web Services," *Proc. of SOSE'08*, Dec. 2008.
[72] T. Berners-Lee, J. Hendler and O. Lassila, "The Semantic Web," *Scientific American,* vol. 284, no.5, pp.34-43, 2001.