# Chapter 8
# A Test Automation Framework for Collaborative Testing of Web Service Dynamic Compositions

**Hong Zhu and Yufeng Zhang**

**Abstract** The dynamic composition of services owned by different vendors demands a high degree of test automation, which must be able to cope with the diversity of service implementation techniques and to meet a wide range of test requirements on-the-fly. These goals are hard to achieve because of the lack of software artefacts of the composed services and the lack of the means of control over test executions and the means of observations on the internal behaviours of composed services. Yet, such integration testing on-the-fly must be non-intrusive and non-disruptive while the composed services are in operation. This chapter presents a test automation framework for such on-the-fly testing of service compositions to facilitate the collaboration between test services through utilisation of Semantic Web Services techniques. In this framework, an ontology of software testing called STOWS are used for the registration, discovery and invocation of test services. The composition of test services is 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. We also demonstrate the uses of the framework by two running examples.

H. Zhu (✉)
Department of Computing and Communication Technologies,
Oxford Brookes University, Oxford OX33 1HX, UK
e-mail: hzhu@brookes.ac.uk

Y. Zhang
National Laboratory for Parallel and distributed Processing School of Computer Science,
The National University of Defense Technology, Changsha, China
e-mail: yufengzhang@nudt.edu.cn

DOI: 10.1007/978-1-4614-7535-4_8,

## 8.1 Introduction

The past few years have seen a rapid growth in the research on testing Web Services (WS) [15, 18], which mostly falls into the following categories.

- **Generation of test cases**. Techniques have been developed to generate test cases from syntax definitions of WS in WSDL [1, 2, 10, 12, 13, 21, 23, 34, 35, 37, 41, 45, 49], business process and behavioural models in BPEL [4, 5, 22, 31, 33, 36, 39, 40, 53], ontology based descriptions of semantics in OWL-S [3, 28, 48], and other formal models of WS such as finite state machines and labelled transition systems [6, 14, 38], grammar graphs [24, 25], and first order logic [46], etc.
- **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 [8] or mock services [27] to replace the other services for testing.
- **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 [19], or a voting mechanism to compare the output from multiple candidate services [44, 47], 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 broken communication connections, and *security* in the environment that is vulnerable to malicious attacks, and so on. A number of prototypes and commercial tools have also been developed to support various activities in testing WS, such as Coyote [45], WS-FIT [37], TAXI [11], PLASTIC [9], LTSA-WS [38]; 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 integration testing [17, 18, 54].

- **The lack of software artefacts**. A service-oriented application commonly consists of services owned by many different stakeholders. Thus, typically, developers of a service have no access to the design document, source code, even the executable code of the other services. These software artefacts are crucial to perform test activities efficiently and effectively.
- **The lack of control over test executions**. A service-oriented application is intrinsically distributed, and typically contains components and services running on hardware owned by other stakeholders. Thus, a tester usually cannot control the test executions of the other owners' services.
- **The lack of a means of observation of internal behaviour**. Another consequence of distributed ownership of services is that testers often cannot observe the internal behaviours of the services owned by other vendors.

Moreover, it is widely recognized that an integration testing technology for WS dynamic composition must meet the following requirements.

- **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.
- **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. A consequence of testing on-the-fly is that it eliminates the possibility of manual testing. Thus, all test activities must be performed automatically.
- **Capability of testing non-intrusively and non-disruptively**. Another 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.

It has been recognized that to address all these issues, testing WS dynamic compositions should be a collaborative effort contributed to by all stakeholders [11, 44, 54]. In this chapter, we present a test automation framework for collaborative testing of web services. The framework presented here has its inception in 2006 [54] based on the author's previous work on agent-based approach to testing web-based systems [55, 56]. A preliminary implementation and case study of the framework was reported in [51]. In [57], the details of test brokers and ontology management were presented and further experiments with the prototype implementation were reported. In [52], the test broker were extended to a general service composition mechanism so that not only test services can be dynamically composed and integrated through service brokers.

The remainder of the chapter is organised as follows. In Sect. 8.2, the framework and its prototype implementation are presented. Section 8.3 illustrates its uses with two running examples in typical scenarios of WS dynamic composition. Section 8.4 discusses its main features and reports the main results of the experiments with the prototype. Finally, Sect. 8.5 concludes the chapter with a discussion of future work.

## 8.2 The Test Automation Framework

This section elaborates the framework and briefly outlines the prototype implementation. More details can be found in [57].

### *8.2.1 The Architecture of the Framework*

As shown in Fig. 8.1, the architecture of the test automation framework consists of

- an ontology of software testing for web services called *STOWS*,
- an ontology manager, which is a web service for the extension and revision of the ontology STOWS, and
- a number of test services.

These components are based on the Semantics Web Service technology and interact with the UDDI and Matchmaker facility.

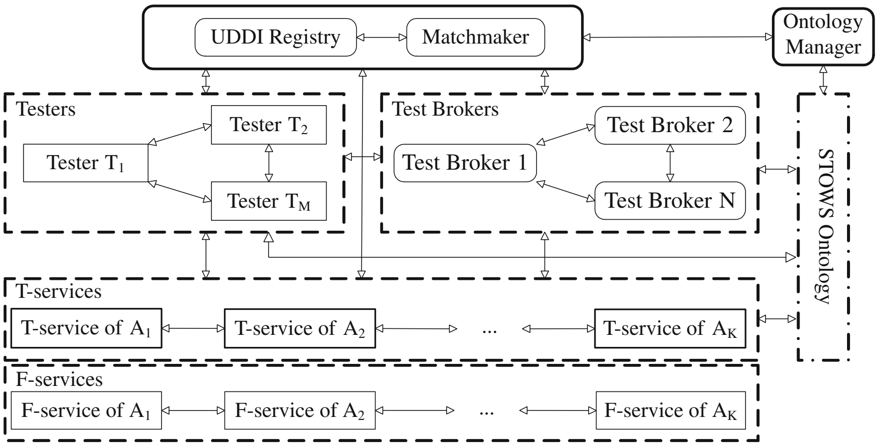


**Fig. 8.1** Reference architecture of the framework

The following subsections will present these components of the architecture.

### *8.2.2 Test Services*

The key notion of the framework is *test services* (*T-service* in short), which are services designated to perform various test tasks [54].

A T-service can be provided by the same organization of the normal service in order to perform the testing of a normal web service. For the sake of clarity, we use *functional service* (or *F-service* in short) to denote the normal services in the sequel.

A Test service can also be provided by a third party that is independent of the normal service provider, and specialized in performing certain testing tasks. A special type of such T-services is test brokers, which coordinate and compose test services in order to perform complicated test tasks.

#### 8.2.2.1 Service Specific T-services

Ideally, each F-service should be accompanied by a special T-service to support the testing of the F-service. Such a T-service should provide the following three types of functions related to testing.

1. *Invoking test execution*. The T-service accompanying an F-service should enable test executions of the F-service to be invoked. Thus, the normal operation of the original F-service is not disturbed by test requests and the cost of testing is 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. A T-service that only provides this test execution function can be regarded as a *mock service* [27]. However, T-service can be much more powerful by providing the following two functions.
2. *Providing required documents*. 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 F-service, the internal design of the F-service such as UML diagrams, 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.
3. *Observing internal behaviour*. Many test activities rely on the information of system's 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.

To ensure that the testing carried out on a T-service faithfully represents the functional services, the following two principles should be maintained 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 the F-service is correct on an input if the T-service passes a test on the input.
- (b) A T-service should have a 'firewall' so that effects on the environment 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 its environment, 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, or an identical copy of the F-service.

For example, the American's Insurance Industry Committee on Motor Vehicle Administration (IICMVA ) requires that each insurance company provides a WS for online verification of car insurances and maintains two identical environments: one for test and one for production [29].

#### 8.2.2.2 General Purpose Testers

Besides the service specific T-service that accompanies an F-service, 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 Sect. 8.1. For the sake of convenience, such general purpose T-services are also called *testers* in the sequel to distinguish them from service specific T-services.

It is worth noting that the framework provides a facility for the integration of testing services rather than any specific testing techniques or tools. Most existing works on WS testing are complementary to the 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 enables test services to provide the software artefacts that testing processes require. The loosely coupled framework lays a foundation for composing various T-services by the utilization of Semantic WS technology.

#### 8.2.2.3 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 Sect. 8.1, 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, but 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.

We have developed a prototype test broker. Figure 8.2 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 passes 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.

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

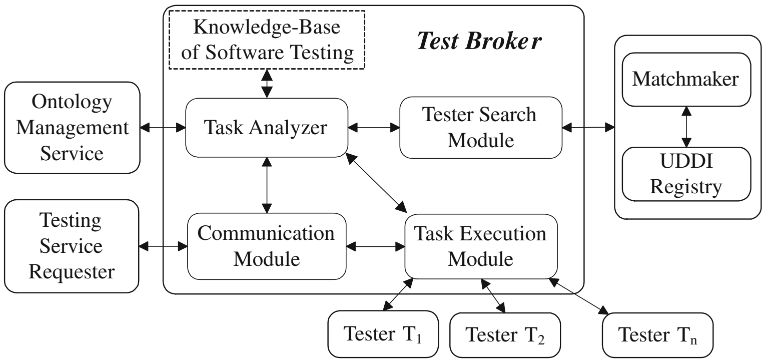


**Fig. 8.2** The structure of a test broker

*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 of test plan executions for each task so that backtracking 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 if any, 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 contains codified knowledge on how a task can be fulfilled by a number of subtasks. Each type of tasks is defined by a set of parameters. There are two kinds of parameters: *descriptive parameters* and *functional parameters*. The former describes the functionality of the task, such as the activity of the task, the execution environment of the task, and so on. The latter gives the data to be transformed by the task, including input and output data. The values of these parameters are concepts defined in the ontology.

The knowledge is represented in the form of rules:

$$T(p_1, \ldots, p_n) \Rightarrow T_1'(p_{1,1}, \ldots, p_{1,n_1}); \ldots; T_k'(p_{k,1}, \ldots, p_{k,n_k})$$

where $T$ is a task and $p_1, \ldots, p_n$ are its parameters. It means that the task $T$ can be decomposed into $k$ subtasks $T_1' \cdots T_n'$, where $p_{i,1}$, ..., $p_{i,n_i} (1 \leq i \leq k)$ are parameters.

It is required that a parameter $p_{i,j}$ of subtask $T_i'$ is constructed from $p_1, \ldots, p_n$ and the output parameters of its previous subtasks, i.e. $\{p_{x,y} | x < i, y \leq n_x\}$. This means that the subtasks can be executed in the order as they occur in the rule. The value of a parameter will be passed from one to the next according to the parameters dependency between subtasks.

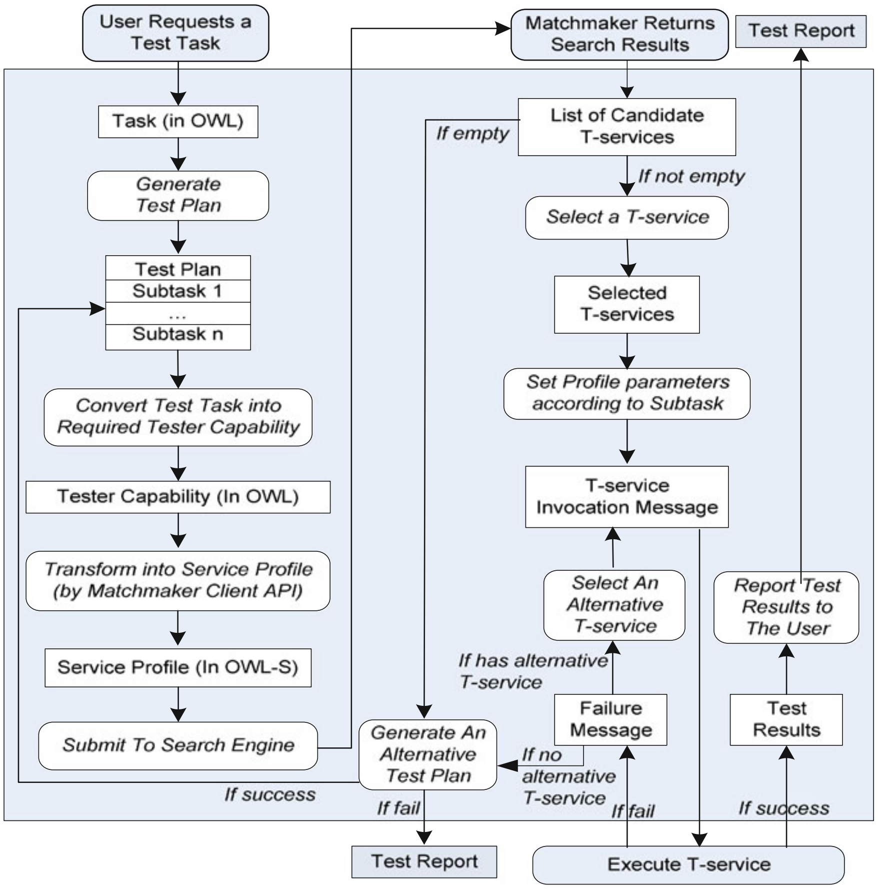
It is also required that each of the output parameters of task $T$ is constructed from the set of output parameters of subtasks $T_i'$ $(i = 1, \ldots, k)$. This is to ensure that task $T$ is realized by the subtasks in the rule.

Therefore, a rule is not only a logic decomposition of a task into several subtasks, but also an expression of the workflow and the collaborations between various kinds of services to complete a specific kind of task. Moreover, from computational point of view, these rules also provide heuristic rules for narrowing the search space for generating service composition plans. In fact, each rule can be considered as a template of test plans. A test task is then checked against the templates one by one. When a match is found, a test plan is produced by instantiating the template. Each rule can also be regarded as a collaboration pattern of T-services with heuristics 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. Thus, the complexity of T-service composition and collaboration can be reduced.

Our implementation of test brokers enables the user to write their own rules and instantiate the knowledge-base so that a number of test brokers can be registered and employed in testing. Figure 8.3 shows the process that the test broker interacts with Matchmaker and other T-services.

### 8.2.3 Registry and Matchmaker

As discussed above, 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 an UDDI registry and a Matchmaker [30].



User Requests a Test Task
Task (in OWL)
Generate Test Plan
Test Plan
Subtask 1
...
Subtask n
Convert Test Task into Required Tester Capability
Tester Capability (In OWL)
Transform into Service Profile (by Matchmaker Client API)
Service Profile (In OWL-S)
Submit To Search Engine
If success
Matchmaker Returns Search Results
Test Report
List of Candidate T-services
If empty
If not empty
Select a T-service
Selected T-services
Set Profile parameters according to Subtask
T-service Invocation Message
Select An Alternative T-service
Report Test Results to The User
If has alternative T-service
Failure Message
Test Results
Generate An Alternative Test Plan
If no alternative T-service
If fail
If fail
If success
Test Report
Execute T-service

**Fig. 8.3** Process model of test broker

The OWL-S/UDDI Matchmaker (Matchmaker for short) extends UDDI registry with a capability based service matching engine [30, 43]. It provides three levels of matching between capability and search request.

1. *Exact matching*: the capabilities in the registry and in the request match exactly.

2. *Plug-in matching*: the service provided is more general than the requested.
3. *Relaxed matching*: the service provided is similar to the requested.

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 [32]. 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.

### 8.2.4 STOWS: Ontology of WS Testing

The semantic information used in the registration, discovery and invocation of T-services are represented in an ontology called STOWS (Software Testing Ontology for WS), which proposed in [54] based on the ontology developed in [55, 56]. 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 formats, etc.

The basic testing concepts include *Tester*, *Artefact*, *Activity*, *Context*, *Method*, and *Environment*. They are described as follows.

- *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 case, test result, 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.

These basic concepts are combined together to express compound testing concepts, which include *Task* and *Capability*.

- *Capability*. 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 of several basic test concepts. The structure of Capability is shown in the UML class diagram given in Fig. 8.4.
- *Task*. 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 as shown in Fig. 8.4, but have different semantics.

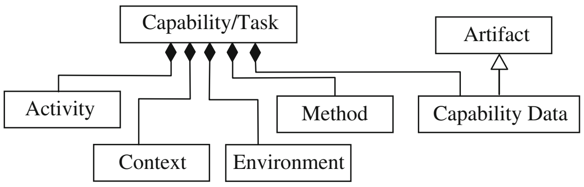


**Fig. 8.4** The structure of capability and task

In OWL-S,[1] semantic descriptions are represented 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. Figure 8.5 shows how the concept of capability is represented in service profile.

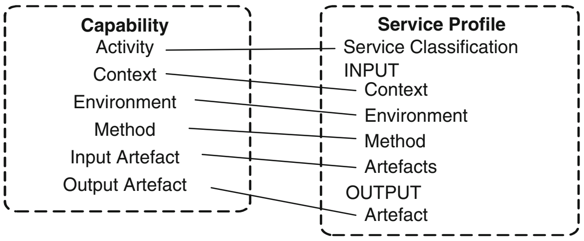


**Fig. 8.5** Mapping between capability and service profile

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. 8.6 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.

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 requests (i.e. testers' capabilities). Similarly, testers' capabilities can be transformed into test subtasks according to the test plan and submitted to the testers. In the implementation of the prototype, we used the Mindswap OWL-S API[2] to parse task and capability profiles and to invoke T-services automatically.

The use of an ontology of software testing provides a standard set of vocabulary for encoding the semantic information passed between T-services as well as for T-service registration and discovery. 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 techniques, methods and tools. Instead, we take the so-called *crowd-sourcing* approach to the construction of the ontology. It is the same approach that Wikipedia is developed. We achieve this by regarding STOWS

[1] http://www.w3.org/Submission/OWL-S/

[2] http://www.mindswap.org/2004/owl-s/api/

```
<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. 8.6** An example of service profile

as an ontology framework in which new vocabulary can be added and updated, and make it open to the public for population. This is supported by a facility for dynamic management of the ontology detailed in the next section.

### *8.2.5 Ontology Manager*

The crowd-sourcing approach to ontology construction is achieved by dynamic management of the ontology through another special service, i.e. the ontology management service (OMS). It provides a mechanism to populate and update the ontology. It is delivered as a WS to facilitate the public access to the mechanism.

The ontology management service is implemented using the Protege-OWL API[3], 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. Therefore, the manipulation of the ontology can be implemented as operations on OWL files. Figure 8.7 shows 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.

[3] http://protege.stanford.edu/plugins/owl/api/guide.html

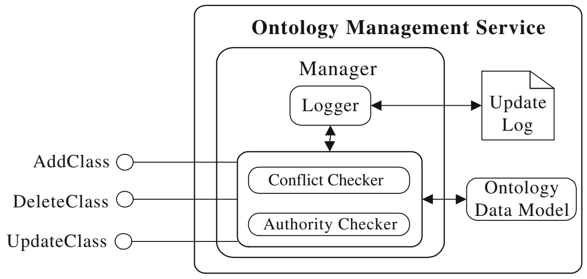


**Fig. 8.7** The structure of OMS

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 TestMethod. 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 SoftwareArtefact. 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 framework 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 an 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 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 it uses 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.

## 8.3 Running Examples

We now illustrate how the framework works in WS integration testing using two running examples of typical scenarios in the dynamic composition of WS.

### *8.3.1 Example 1: Testing On-The-Fly for WS Dynamic Composition*

Our first example is the integration testing in the dynamic composition of the services of a car insurance broker with the web services of an insurance company.

#### 8.3.1.1 The Scenario

Suppose that a fictitious car insurance broker CIB operates 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, including its usage, and the payment. It uses the WS of its bank B to check the validity of user's payment information, passes 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. 8.8.

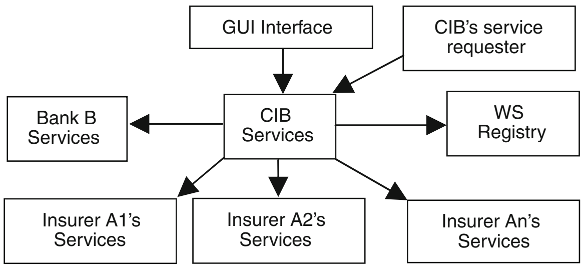


**Fig. 8.8** Structure of car insurance broker services

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. Here, we focus on the integration with dynamic binding. Thus, suppose that CIB will dynamically compose with the WS of the PingAn Insurance Ltd. in China that provides car insurance to the customers through a web-based application.[4] It is a real-world example.

#### 8.3.1.2 Architecture of Test Services

By applying the framework to the scenario, each of the functional WS of the bank B, CIB and insurer $A_i$ has an accompanying T-service. Thus, we have the following architecture shown in Fig. 8.9. In particular, the following services are involved in the testing of the dynamic composition of CIB and the WS of PingAn Insurance.

- CIQS: the WS of the PingAn Insurance. It is the web service to be tested.
- TCE: a service specific T-service that executes the test cases for CIQS.
- TCG: a special purpose WS testing tool that generates test cases.
- CIB: the WS of the car insurance broker CIB. It acts as the testing requester, and generates and submits test tasks to the test broker to test CIQS.

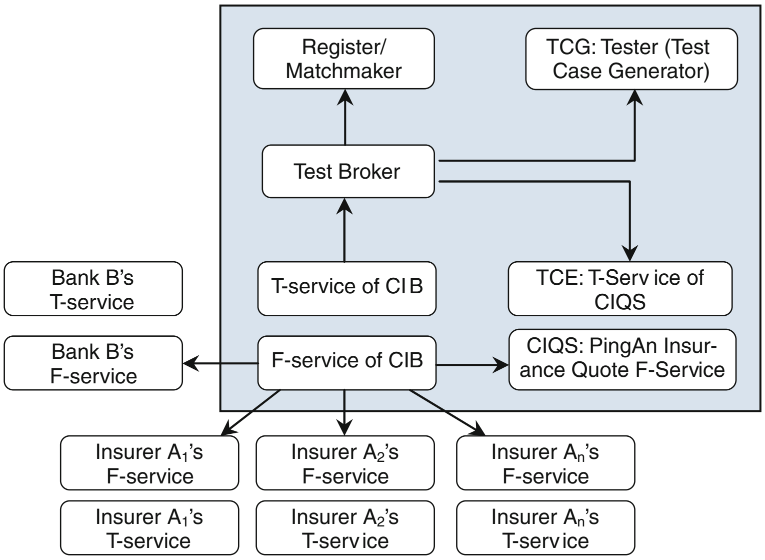

Register/ Matchmaker
TCG: Tester (Test Case Generator)
Test Broker
Bank B's T-service
T-service of CIB
TCE: T-Service of CIQS
Bank B's F-service
F-service of CIB
CIQS: PingAn Insurance Quote F-Service
Insurer $A_1$'s F-service
Insurer $A_2$'s F-service
Insurer $A_n$'s F-service
Insurer $A_1$'s T-service
Insurer $A_2$'s T-service
Insurer $A_n$'s T-service

**Fig. 8.9** System architecture of the typical scenario

These T-services are registered to the UDDI registry using the STOWS ontology. For example, TCG is a WS that takes the WSDL file of a service to be tested as input

[4] http://www.pingan.com/campaign/channels/pingan/car-quote/index.jsp

and generates random test cases as output. These artefacts are stored in files and referred to through URLs of the file locations. To describe this service, the following classes were added into the ontology.

- WSDL: a subclass of ServiceDescription, which is in turn the subclass of Artefact. It stands for the WSDL document of a service.
- ServiceTesting: a subclass of Context that stands for service testing.
- RandomTestingMethod: a subclass of Method that stands for the random testing method in test case generation.
- CarInsuranceQuoteServiceTestCase: a subclass of TestCase that stands for the test case file for testing car insurance quote service.

In the service profile that describes the capability of TCG, the serviceClassification is TestCaseGeneration. The Input artefact is WSDL. The context of TCG is ServiceTesting. Its environment is of type Environment, which is the ancestor of all the classes and stands for test environments. This means it imposes no specific requirement on the environment. Its method is RandomTestingMethod. The output artefact is of type CarInsuranceQuoteServiceTestCase.

#### 8.3.1.3 Collaboration Process

Consider the situation that the CIB intends to establish a dynamic composition with insurer PingAn and to test the service on-the-fly. It delegates the task to a test broker TB. Figure 8.10 shows a typical example of collaboration processes managed by TB.

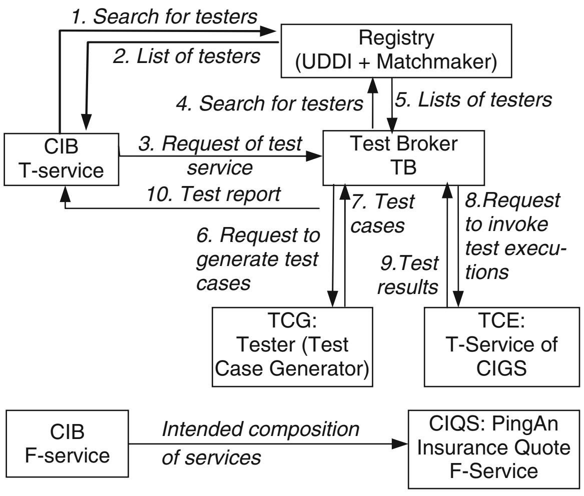

1. Search for testers
2. List of testers
Registry (UDDI + Matchmaker)
4. Search for testers
5. Lists of testers
CIB T-service
3. Request of test service
Test Broker TB
10. Test report
7. Test cases
8.Request to invoke test executions
6. Request to generate test cases
9.Test results
TCG: Tester (Test Case Generator)
TCE: T-Service of CIGS
CIB F-service
Intended composition of services
CIQS: PingAn Insurance Quote F-Service

**Fig. 8.10** The collaboration process in a typical scenario

The process starts with the generation of a test task by CIB's WS and submission of a search request for finding a proper tester to the service registry. The search request message contains a test task, which is matched against the capabilities of the registered testers. The search result is a list of testers who are capable of performing the task. From this list, the test broker TB is selected. A test request as shown in Fig. 8.11 is then sent to TB requesting to test CIQS.

Once the test broker receives the test task, it generates a test plan that consists of two subtasks:

- *Subtask 1*: Generating test cases according to a car insurance industry standard. The input artefact of the task is of type WSDL. The output of this subtask is of type CarInsuranceQuoteServiceTestCase.
- *Subtask 2*: Executing test cases and reporting test results. Its input is of type CarInsuranceQuoteServiceTestCase and its output type is CarInsuranceQuoteServiceTestResult.

```
<Task rdf:ID="insuranceQuoteServiceTestingTask">
    <needContext>
        <ServiceTesting rdf:ID="serviceTesting"/>
    </needContext>
    <needMethod>
      <RandomTestingMethod rdf:ID="randomTestingMethod"/>
    </needMethod>
    <needEnvironment>
        <Environment rdf:ID="environment"/>
    </needEnvironment>
    <needServiceClassification>
        <ServiceClassification rdf:ID="serviceClassification"/>
    </needServiceClassification >
    <inputArtefact>
        <WSDL rdf:ID="CarInsuranceQuoteServiceWSDL">
            <Location rdf:datatype="http://www.w3.org/2001/XMLSchema#string ">
                http://.../CarInsuranceQuoteService?wsdl
            </Location>
        </WSDL >
    </inputArtefact>
    <outputArtefact>
        <CarInsuranceQuoteServiceTestResult rdf:ID="testresult">
            <Location rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
                http://.../artefacts/testresult/fictitioustestresult.xml
            </Location>
        </CarInsuranceQuoteServiceTestResult >
    </outputArtefact>
</Task>
```

**Fig. 8.11** An instance of test tasks in the running example

For each subtask in the test plan, the broker translates the subtask into the corresponding capability description and constructs a service profile. The test broker then submits the service profile to the Matchmaker to search for appropriate testers. In this case, testers TCG and TCE are selected for the subtasks, respectively. The test planning finished with each subtask associated with a tester, and the test plan is passed to the execution module of the test broker for executing the subtasks.

The task execution module of the test broker calls the testers associated to each subtask according to the order given in the test plan. Data are passed from one subtask to another through invocation messages. In particular, the output artefact of the first subtask is passed to the second subtask. The output of the second subtask is the final result of the test, which is an OWL object. It is then returned to the client.

## *8.3.2 Example 2: Wrapping A Testing Tool into a Test Service*

In this running example we demonstrate how to wrap an automated testing tool into a test service and how the tester can be composed together with other T-services to accomplish complex testing tasks.

### 8.3.2.1 Wrapping a Testing Tool

The testing tool in this running example is a general purpose testing tool called CASCAT [50], which generates test cases from algebraic specifications. It is wrapped into a web service by providing it with a WS interface. The web service version of the tool is referred to as TCG in the sequel. The following gives some technical details of the registration, search and invocation of the tester.

In the registration of TCG, 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 of the file locations. To describe this service, the following new classes are 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 Environment and represents no requirement on the test environment. Its method is CASOCCmethod. The output artefact is TestCase.

#### 8.3.2.2 Collaboration Process

Similar to the first running example, suppose that a client wants to test a WS called NCS, which is a web service that provides numeric calculations of complex numbers. The client constructs a test task and submits it to the registry to search for a tester. As a result, a test broker is found to perform the testing.

Figure 8.12 shows the test task that client 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.

Once the test broker receives the test task, it decomposes it into subtasks and generated a test plan that consisted of the following 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 the T-service of NCS. 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.

```
<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. 8.12** The task to test NCS based on algebraic specification

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 submits the service profile to the Matchmaker to search for appropriate testers. In this case, testers TCG, TFT and T-NCS are discovered for the subtasks, respectively. The test planning finishes with each subtask associated with a tester. The test plan is then passed to the execution module for executing the subtasks.

The task execution module calls the testers associated to each subtask according to the order given in the test plan. Data are passed from one subtask to another by the construction of invocation message to the testers. In particular, the output artefact of a subtask is passed to the next subtask. The output of the third subtask is the final result of the test, which is again an OWL object. It is returned to the client by the broker. Figure 8.13 summarises the collaboration process described above.

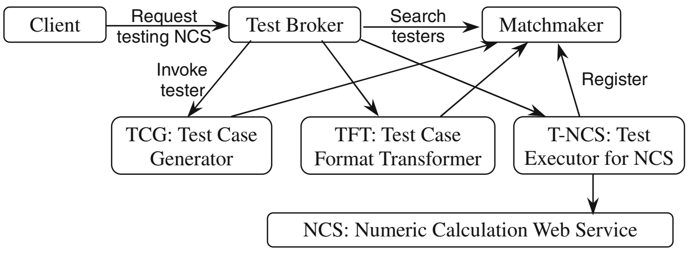


**Fig. 8.13** The collaboration between the web services in running Example 2

## 8.4 Discussion: Main Feature of the Framework

The framework implements collaborative testing of WS within the 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. It does not only applicable to functional testing as demonstrated in the running examples, but also applicable to non-functional tests, for example, through collaboration with a non-functional test service. The collaborations between test services are autonomous rather than enforced. That is, what to test and how to test is the choice of the service requester, but 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. Test brokers are also T-services but specialised in the composition of T-services. 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.

**A. Scalability**

The framework is scalable since T-services are distributed and there is no extra-burden on UDDI servers. Experiments reported in [52, 57] shows that the average test service search time increases with the number of testers in the registry, but in almost a linear manner, as shown in Fig. 8.14. With the size of knowledge base increases, the time spent by a test broker to generate test service composition plans also increases, but again in an almost linear rate as shown in Fig. 8.15. With the increase of the complexity of testing tasks, which is measured by the number of different types of subtasks to fulfil the task, the time overhead increases in a quadratic polynomial function as shown in Fig. 8.16. Therefore, the test 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.

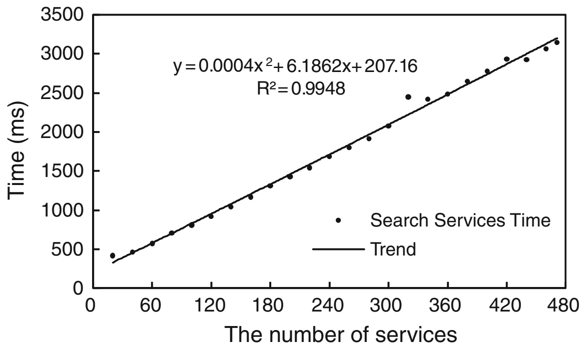


**Fig. 8.14** Execution time dependence on the number of testers registered in UDDI

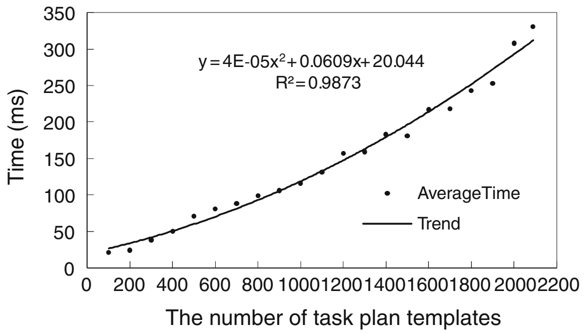


**Fig. 8.15** Execution time dependence on the number of plan templates in knowledge-base

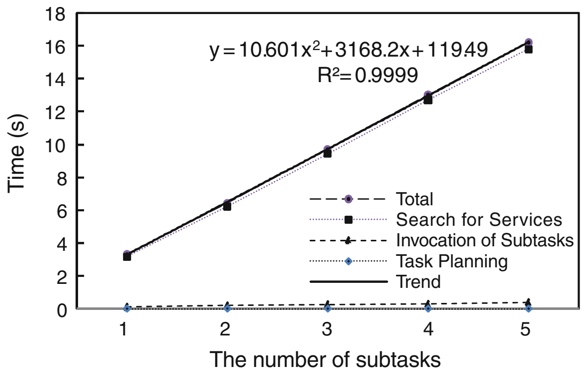


**Fig. 8.16** Execution time dependence on the number of subtasks

### B. Feasibility

The framework is implemented without any change to the existing standards of Semantic WS [7]. A case study reported in [57] demonstrated that a wide range of different types of test services can be supported and integrated into the framework. Table 8.1 summarises the services used in the case study.[5]

### C. Capability of dealing with diversity

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. An experiment applying data mutation testing techniques [42] shows clearly that the framework is capable of dealing with services of subtle differences so that the best match can be automatically selected to perform testing tasks [57].

### D. Fully automated for testing on-the-fly

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.

### E. Extendibility

This framework employs an ontology management facility to enhance its extendibility. With this, the software testing ontology can be extended and maintained through public services.

[5] Java NCSS can be found at URL: http://javancss.codehaus.org/, and PMD can be found at URL: http://pmd.sourceforge.net/

**Table 8.1** Testers integrated in the framework

| Name | Description |
|---|---|
| CASCAT [50] | A CASOCC-based test case generation tool |
| Test case format translator | Translates 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 [16] | Generate and execute test cases from C source code by symbolic execution |
| Magic [20] | Check conformance between component specifications and their implementations |
| XML comparator | Compare XML files |
| Java NCSS | Measure two standard metrics for Java program |
| Findbugs [26] | Find bugs in Java program by static analysis |
| PMD | A static analysis tool for finding potential bugs and other problems in Java source code |
| WSDL-based test generator [2] | A WSDL based test case generation tool |
| Web service test case executor [2] | Execute the test case generated by WSDL based test case generator |

## 8.5 Conclusion and Future work

In this chapter, we presented a 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 its scalability.

The test broker in the framework plays an important role in automation of testing processes. Further research on the design and implementation of 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. Currently, such knowledge of software testing process is represented in the form of task decomposition rules. A question is whether such knowledge can be encoded in a process definition language such as BPEL. If yes, a careful analysis of the benefit and comparison of the two are necessary. Another direction to enhance the functionality of test brokers is to associate monitoring functions to brokers as Tsai et al. suggested 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 in the prototype 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.

Moreover, as discussed in Sect. 8.1, a particular difficulty in testing WS is due to the lack of software artefacts to support test activities. The framework presented in this chapter 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. Consistency problem may occur when the ontology is updated during the execution of a task. 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.

**Acknowledgments** The work reported in this chapter is partly funded by the National Basic Research Program of China (973) under Grant No. 2011CB302603 and the National Natural Science Foundation of China under Grant No. 60725206.

## References

1. de Almeida, L.F., Vergilio, S.R.: Exploring perturbation based testing for web services. In: Proc. of ICWS'06, pp. 717–726. IEEE CS (2006)
2. Bai, X., Dong, W., Tsai, W., Chen, Y.: Wsdl-based automatic test case generation for web services testing. In: Proc. of SOSE'05, pp. 215–220. IEEE CS (2005)
3. Bai, X., Lee, S., Tsai, W.T., Chen, Y.: Ontology-based test modeling and partition testing of web services. In: Proc. of ICWS'08, pp. 465–472. IEEE CS (2008)
4. Bartolini, C., Bertolino, A., Marchetti, E.: Introducing service-oriented coverage testing. In: Proc. of ASE'08, pp. 57–64. IEEE CS (2008)
5. Bartolini, C., Bertolino, A., Marchetti, E., Parissis, I.: Data flow-based validation of web services compositions: Perspectives and examples. In: R.e.a. Lemos V (ed.) Architecting Dependable Systems, LNCS, vol. 5135, pp. 298–325. Springer-Verlag (2008)
6. Belli, F., Linschulte, M.: Event-driven modeling and testing of web services. In: Proc. of COMPSAC'08, pp. 1168–1173. IEEE CS (2008)
7. Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American **284**(5), 34–43 (2001).
8. Bertolino, A., Angelis, G.D., Frantzen, L., Polini, A.: Model-based generation of testbeds for web services. In: Proc. of TestCom/FATES'08, pp. 266–282 (2008)
9. Bertolino, A., Angelis, G.D., Frantzen, L., Polini, A.: The plastic framework and tools for testing service-oriented applications. In: Software Engineering: Int'l Summer Schools, (ISSSE'08), pp. 106–139 (2008)

10. Bertolino, A., Gao, J., Marchetti, E.: Xml every-flavor testing. In: Proc. of WEBIST'06, pp. 268–273. INSTICC Press (2006)
11. Bertolino, A., Gao, J., Marchetti, E., A.Polini: Taxi-a tool for xml-based testing. In: Proc. of ICSE'07 (Companion), pp. 53–54. IEEE CS (2007)
12. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Automatic test data generation for xml schema-based partition testing. In: Proc. of AST'07, p. 4. IEEE CS (2007)
13. Bertolino, A., Gao, J., Marchetti, E., Polini, A.: Systematic generation of xml instances to test complex software applications. In: N.e.a. Guelfi (ed.) Rapid Integration of Software Engineering Techniques, *LNCS*, vol. 4401, pp. 114–129. Springer (2007)
14. Bertolino, A., Polini, A.: The audition framework for testing web services interoperability. In: Proc. of EUROMICRO'05, pp. 134–142 (2005)
15. Bozkurt, M., Harman, M., Hassoun, Y.: Testing & verification in service-oriented architecture: A survey. Software Testing, Verification and Reliability (STVR) (To Appear).
16. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI (2008)
17. Canfora, G., Penta, M.: Testing services and service-centric systems: Challenges and opportunities. IT Professional **8**(2), 10–17 (2006)
18. Canfora, G., Penta, M.: Service-oriented architectures testing: A survey. In: A. Lucia, F. Ferrucci (eds.) Software Engineering: Int'l Summer Schools (ISSSE 2006–2008), Revised Tutorial Lectures, *LNCS*, vol. 5413, pp. 78–105. Springer-Verlag (2009)
19. Chan, W.K., Cheung, S.C., Leung, K.R.P.H.: A metamorphic testing approach for online testing of service-oriented software applications. Int'l Journal of Web Services Research **4**(2), 61–81 (2007)
20. Edmund, S.C., Clarke, E., Groce, A., Jha, S., Vienna, T.: Modular verification of software components in c. IEEE Trans. Softw. Eng. **30**, 388–402 (2004)
21. Emer, M.P., Vergilio, S.R., Jino, M.: A testing approach for xml schemas. In: Proc. of COMPSAC'05, pp. 57–62. IEEE CS (2005)
22. Garcia-Fanjul, J., Tuya, J., de la Riva, C.: Generating test cases specifications for bpel compositions of web services using spin. In: Proc. of WS-MaTe (2006)
23. Hanna, S., Munro, M.: An approach for wsdl-based automated robustness testing of web services. In: C.e.a. Barry (ed.) Information Systems Development: Challenges in Practice, Theory, and Education, vol. 2, pp. 493–504. Springer (2009)
24. Heckel, R., Lohmann, M.: Towards contract-based testing of web services. Electronic Notes in Theoretical Computer Science **82**(6) (2004)
25. Heckel, R., Mariani, L.: Automatic conformance testing of web services. In: Proc. of FASE'05, pp. 34–48. Springer (2005)
26. Hovemeyer, D., Pugh, W.: Finding more null pointer bugs, but not too many. In: Proc. of PASTE'07, pp. 9–14 (2007)
27. Huang, H., Liu, H., Li, Z., Zhu, J.: Surrogate: A simulation apparatus for continuous integration testing in service oriented architecture. In: Proc. of SCC'08, vol. 2, pp. 223–230. IEEE CS (2008)
28. Huang, H., Tsai, W., Paul, R., Chen, Y.: Automated model checking and testing for composite web services. In: Proc. of ISORC'05, pp. 300–307. IEEE CS (2005)
29. IICMVA: Model user guide for implementing online insurance verification, version 4, Insurance Industry Committee on Motor Vehicle Administration, USA. http://www.iicmva.com/IICMVAPublications.html (2010). (Accessed on 20 Oct. 2010).
30. K. Sycara M. Paolucci, A., Srinivasan, N.: Automated discovery, interaction and composition of semantic web services. J. Web Semantics **1**(1), 27–46 (2003)
31. Kaschner, K., Lohmann, N.: Automatic test case generation for services. In: Proc. of Fourth Int'l Workshop on Engineering Service-Oriented Applications: Analysis and Design (WESOA 2008), LNCS. Springer-Verlag (2008)
32. Kawamura, T., Blasio, J.A.D., Hasegawa, T., Paolucci, M., Sycara, K.: A preliminary report of a public experiment of a semantic service matchmaker combined with a uddi business registry. In: Proc. of ICSOC'03, pp. 208–224. IEEE CS (2003)

33. Lallali, M., Zaidi, F., Cavalli, A., Hwang, I.: Automatic timed test case generation for web services composition. In: Proc. of ECOWS'08, pp. 53–62 (2008)
34. Lee, S.C., Offutt, J.: Generating test cases for xml-based web component interactions using mutation analysis. In: Proc. of ISSRE'01, pp. 200–209. IEEE CS (2001)
35. Li, J.B., Miller, J.: Testing the semantics of w3c xml schema. In: Proc. of COMPSAC'05, pp. 443–448. IEEE CS (2005)
36. Li, Z., Sun, W., Jiang, Z.B., Zhang, X.: Bpel4ws unit testing: Framework and implementation. In: Proc. of ICWS'05, pp. 103–110. IEEE CS (2005)
37. Looker, N., Munro, M., Xu, J.: Ws-fit: A tool for dependability analysis of web services. In: Proc. of COMPSAC'04, pp. 120–123. IEEE CS (2004)
38. Magee, J., Kramer, J., Uchitel, S., Foster, H.: Ltsa-ws: a tool for model-based verification of web service compositions and choreography. In: Proc. of ICSE'06, pp. 771–774. IEEE CS (2006)
39. Mayer, P.: Design and implementation of a framework for testing bpel compositions. Master's thesis, Leibnitz Univ., Germany (2006)
40. Mei, L., Chan, W.K., Tse, T.H.: Data flow testing of service-oriented workflow applications. In: Proc. of ICSE'08, pp. 371–380. IEEE CS (2008)
41. Offutt, J., Xu, W.: Generating test cases for web services using data perturbation. SIGSOFT Softw. Eng. Notes **29**(5), 1–10 (2004)
42. Shan, L., Zhu, H.: Generating structurally complex test cases by data mutation. The Computer Journal **52**, 571–588 (2009)
43. Srinivasan, N., Paolucci, M., Sycara, K.: Adding owl-s to uddi, implementation and throughput. In: Proc. of The 1st Int'l Workshop on Semantic Web Services and Web Process Composition, pp. 169–182 (2004)
44. Tsai, W., Chen, Y., Paul, R., Liao, N., Huang, H.: Cooperative and group testing in verification of dynamic composite web services. In: Proc. of COMPSAC'04, vol. 2: Workshops and Fast Abstracts, pp. 170–173. IEEE CS (2004)
45. Tsai, W., Paul, R., Song, W., Cao, Z.: Coyote: An xml-based framework for web services testing. In: Proc. of HASE'02, pp. 173–174. IEEE CS (2002)
46. Tsai, W., Wei, X., Chen, Y., Paul, R., Bai, X.: Swiss cheese test case generation for web services testing. IEICE - Trans. Inf. Syst. **88**(12), 2691–2698 (2005)
47. Tsai, W., Zhou, X., Chen, Y., Bai, X.: On testing and evaluating service-oriented software. Computer **41**(8), 40–46 (2008)
48. Wang, Y., Bai, X., Li, J., Huang, R.: Ontology-based test case generation for testing web services. In: Proc. of ISADS'07, pp. 43–50. IEEE CS (2007)
49. Xu, W., Offutt, J., Luo, J.: Testing web services by xml perturbation. In: Proc. of ISSRE'05, pp. 257–266. IEEE CS (2005)
50. Yu, B., Kong, L., Zhang, Y., Zhu, H.: Testing java components based on algebraic specifications. In: Proc. of ICST'08, pp. 190–199. IEEE CS (2008)
51. Zhang, Y., Zhu, H.: Ontology for service oriented testing of web services. In: Proc. of SOSE'08. IEEE CS (2008)
52. Zhang, Y., Zhu, H.: An intelligent broker approach to semantics-based service composition. In: Proc. of COMPSAC 2011, pp. 20–25. IEEE CS, Munich, Germany (2011)
53. Zheng, Y., Zhou, J., Krause, P.: An automatic test case generation framework for web services. Journal of Software **2**(3), 64–77 (2007)
54. Zhu, H.: A framework for service-oriented testing of web services. In: Proc. of COMPSAC'06, pp. 679–691. IEEE CS (2006)
55. Zhu, H., Huo, Q.: Developing a software testing ontology in uml for a software growth environment of web-based applications. In: e. H. Yang (ed.) Software Evolution with UML and XML, pp. 263–295. IDEA Group Inc. (2005)
56. Zhu, H., Huo, Q., Greenwood, S.: A multi-agent software environment for testing web-based applications. In: Proc. of COMPSAC'03, pp. 210–215. IEEE CS (2003)
57. Zhu, H., Zhang, Y.: Collaborative testing of web services. IEEE Transactions on Services Computing **5**(1), 116–130 (2012)