Ontology-Based Test Modeling and Partition Testing of Web Services

Xiaoying Bai and Shufang Lee
Department of Computer Science and Technology,
Tsinghua University, Beijing, China
baixy@tsinghua.edu.cn, lisf06@mails.tsinghua.edu.cn

Wei-Tek Tsai and Yinong Chen
Computer Science & Engineering Department,
Arizona State University, AZ, USA
{wtsai, yinong}@asu.edu

Abstract

Testing is useful to establish trust between service providers and clients. To test the service-oriented applications, automated and specification-based test generation and test collaboration are necessary. The paper proposes an ontology-based approach for Web Services (WS) testing. A Test Ontology Model (TOM) is defined to specify the test concepts, relationships, and semantics from two aspects: test design (such as test data, test behavior, and test cases) and test execution (such as test plan, schedule and configuration). The TOM specification using OWL (Web Ontology Language) can serve as test contracts among test components. Based on the WS semantic specification in OWL-S, the paper discusses the techniques to generate the sub-domains for input partition testing. Data pools are established for each parameter of the specified service. Data partitions are derived by class property and relationship analysis. Completeness and consistency (C&C) checking can be performed on the data partitions and data values, both within the TOM and against the OWL-S, by ontology class computation and reasoning. A prototype tool is implemented to support OWL-S analysis, test ontology generation and C&C checking.

Keywords: Test Model, partition testing, ontology, Web services, OWL-S

1   Introduction

Service-Oriented Architecture (SOA) and its implementation Web Services (WS) introduce an open platform that enables online service registration, discovery, binding, composition, and other dynamic functions based on standard Internet protocols. Dependability has been a critical obstacle to the service-oriented applications because the service clients may not know the service providers and thus may not trust the services dynamically discovered by the service broker. Automated testing is therefore necessary throughout the dynamic process to enforce Verification and Validation (V&V) of the service dependability properties such as correctness, performance and reliability. However, the shift from traditional product-oriented development to the service-oriented computing invalidates many traditional software testing techniques [4][6][10].

This paper is motivated by the two issues of WS testing. The first is the contract-based collaborative testing. Different from the traditional software development, service providers, clients, and brokers in SOA may be independent of each other. The service provider may not foresee all the usage scenarios in various composition contexts. Thus, running a service with a specific scenario outside the intended context may trigger a failure. On the other side, a service client may make certain implicit assumptions of the service such as pre-conditions and post-conditions which may not be explicitly defined in the service interface. It is necessary that all the stakeholders join the test collaboration by sharing their test knowledge, assumptions, design, results, and fault models. A testing contract enables such dynamic test collaboration.

The second issue is of automated test case generation based on the service specifications. In SOA, services may establish their collaborations at runtime by negotiating through specifications, including interface operations, composition workflow, and service-level agreement. Testing needs to be automated and incorporated into the process to support runtime V&V of the services and composite services at various levels such as unit testing, interface testing, integration testing, and collaboration testing. Thus, specification-based test generation is by nature part of the service-oriented application development. However, the intelligence of the test case generator is limited by the information represented in the specification language. In addition to the syntax and structures, it is also important to derive test cases based on service operation and composition semantics such as dependencies and constraints.

To address the first issue, a contract-based collaborative V&V framework was proposed [1][18]. Testing contract specifies the key test activities such as test data, test cases and test schedule. It defines the collaboration agreement among test components [3] as well as all the stakeholders involved in service testing. The paper proposed an ontology model for test contract definition and specification called TOM (Test

Ontology Model). The ontology technique enables semantic definition of the test artifacts using classes, properties, relationships and constraints. TOM defines the ontology for test design and execution. An OWL (Web Ontology Language) [21] specification is constructed to specify TOM.

To address the second issue, one needs to generate test cases/script based on the semantic specifications. Particularly, WS semantic specification OWL-S [22] is used, which is a XML-based ontology description language to specify service composition semantics, including inputs, outputs, preconditions and effects (collectively called IOPE). This paper proposes test input generation for input partition testing based on service input ontology using a testing framework proposed in [2][9][17][19].

Figure 1 shows the overall approach. The service semantic specification (e.g., in OWL-S) was taken as an input to the testing system. Test cases are automatically generated by the test generator. The generated test artifacts are specified by the TOM OWL specification, which serves as the contracts among test components, including:

- Test generator, which parses the WS specifications and generates test cases encoded in TOM;
- Test master, which organizes test plan, schedules test tasks, and coordinates test execution; and
- Test agents, which mobiles and exercises the test cases on the services.

The Ontology Infrastructure provides domain ontology information for both service application domain and test strategies, facilitates ontology editing and analysis, and improves the intelligence of test generation with rules and reasoning techniques.
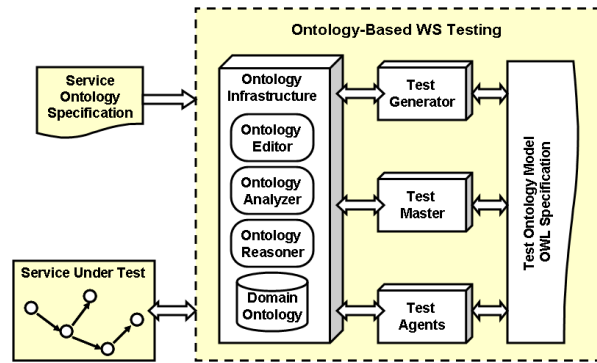

Figure 1 Ontology-based WS Testing

The rest of the paper is organized as follows. Section 2 introduces the example TravelSearch service and its ontology to be used by other sections. Section 3 presents the TOM. Section 4 elaborates the test case generation process and Completeness and Consistency (C&C) checking on the data partitions. Section 5

discusses the prototype tool implementation and experiments performed. Section 6 compares with the related works, including WS testing and partition testing. Finally, section 7 concludes this paper.

2 Example Travel Ontology

The ontology of a service called TravelSearch is specified using OWL-S as shown in Figure 2. The service takes inputs *Activity* and *Accommodation*, and returns *Destination*.

```
<process:AtomicProcess rdf:ID="TravelSearchProcess">
    <service:describes rdf:resource="#TravelSearchService"/>

    <process:hasInput rdf:resource="#Activity"/>
    <process:hasInput rdf:resource="#Accomodation"/>
    <process:hasOutput rdf:resource="#Destination"/>
</process:AtomicProcess>

<process:Input rdf:ID="Activity">
    <process:parameterType
    rdf:datatype="&xsd;#anyURI">&bibtex;#Activity</process:parameterType>
</process:Input>

<process:Input rdf:ID="Accomodation">
    <process:parameterType
    rdf:datatype="&xsd;#anyURI">&bibtex;#Accomodation</process:parameterType>
</process:Input>

<process:Output rdf:ID="Destination">
    <process:parameterType
    rdf:datatype="&xsd;#anyURI">&concepts;#Destination</process:parameterType>
</process:Output>
```
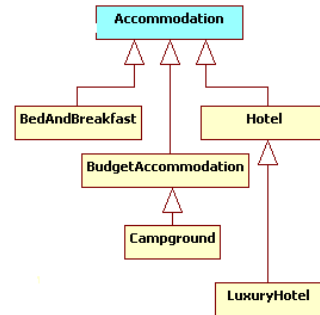Figure 2 OWL-S specification of TravelSearch service

Figure 3 Example of the Accommodation ontology

Figure 3 shows the *Accommodation* definition. The *Accommodation* has three subclasses including *Hotel*, *BedAndBreakfast*, and *BudgetAccommodation*. *Budget Accommodation* has one subclass *Campground* and *Hotel* has one subclass *LuxuryHotel*. The *Accommodation* has a property *hasRating* which can have only three values: *OneStar, TwoStar* and *ThreeStar*. In addition, *Accommodation* has the following constraints:

1. *BudgetAccommodation* are those with one or two star ratings, i.e.,
   $BudgetAccommodation \equiv (Accommodation \cap hasRating(OneStar \cup TwoStart))$.

2. *Campground* are those with one star ratings, i.e.,
   $Campground \subset (Accommodation \cap hasRating(OneStar))$.

3. *LuxuryHotel* are those with three star ratings, i.e., *LuxuryHotel ⊆ (Hotelhas ∩ Rating(ThreeStar)).*

3   Test Ontology Model

A test model provides a standard definition and description of test information such as the organizational structure of test suites, test data definition, test plan, and scheduling. This model should be specified independent of the platform and programming language of the SUT (Subject Under Test), and thus can be easily maintained and reused. A typical test model is the UML (Unified Modeling Language) U2TP (UML 2.0 Test Profile) [23]. U2TP provides a language for modeling testing, including test architecture, behavior, data, and time. It uses the UML meta-modeling approach. A MOF (Meta Object Facility) meta-model is defined to enable the use of U2TP independent of UML.

This paper proposes an ontology-based test model TOM that is compatible to the U2TP. In addition, it enriches the semantics of the U2TP model with class properties and constraints using ontology information. Furthermore, it provides better supports for testing by enabling C&C checking to detect the test design faults of incompleteness, inconsistency, and duplication.
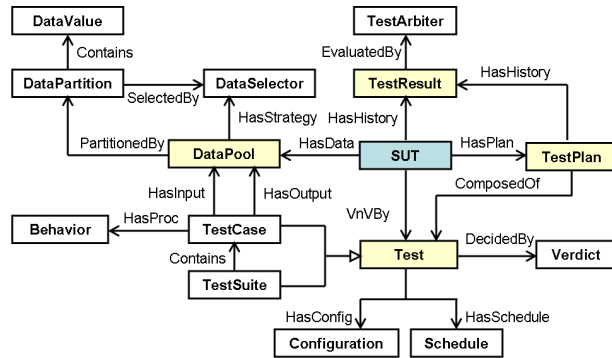

Figure 4 Test Ontology Model

TOM specifies test artifacts from two perspectives: test *design* and *execution*. By test design, it specifies the key concepts of test cases and test suites from two aspects: 1) Test data specifies the data related concepts including data pool, data partitions, data selection strategy and data values for each of the input/output defined in the service; and 2) Test behavior specifies the preparation, test procedures, and expected results for each test to be exercised. By test execution, it specifies the information for exercising the designed test cases/suites. It also contains two parts: 1) Test plan, which specifies the organization, scheduling, and configuration of the distributed execution of test cases; and 2) Test results which specifies the collected test results for statistical analysis and quality evaluation.

Figure 4 shows the concepts and relationship among the modeling elements in the TOM. As shown in the diagram, each service (SUT) is associated with a group of test cases, a set of data pools carrying the test data, a set of test plans for scheduling the test execution, and a set of test results corresponding to each test execution.

A test case has two parts: the data part and the behavior part. A data pool is the container of test data for a specific testing purpose. For example, a service may have three pools for all its data, including functional testing, reliability testing, and security testing respectively. The pool for a data is further divided into data partitions representing test sub-domains. The data can be selected from different partitions at runtime by the data selectors with various data selection criteria. For example, taking the TravelSearch example, the input parameter *accommodation* may have a data pool for functional testing, which can be partitioned into three categories by its ratings as one-star, two-star and three-star rating accommodations. Thus it may be useful for a regression testing "only re-test the search service for three-star accommodations". The behavior part defines the actions and orders of actions for a test case. The actions maybe test preparation, event triggering, data input, and test cleanup. The model enables the dynamic association between the actions and its associated data.

A test plan defines the organization of test executions. Test cases can be organized into test suites. The test plan defines the scheduling of test executions, such as sequential and concurrency. It also defines the configuration of a test run, such as the deployment of test agents in a LAN for distributed testing. The test plan enables the dynamic re-composition of test cases and re-configuration of test executions.

TOM can support the collaboration between all the stakeholders such as service providers, brokers, clients, and other independent service evaluators [1][18], and they all can contribute and share the testing assets including test case, test data and test results following the Web 2.0 principle where a user can be an active contributor.

4   Ontology-Based Data Partition Generation

A major issue to automated testing is the generation of input data. Partition testing has been used with many testing methods, including white-box and black-box testing [7][8][15]. Parameters, such as inputs or environment, are classified into sub-domains called partitions. Test data are selected from the partitions based on coverage strategies. The performance of partition testing heavily depends on the adequacy of selected partitions. However, traditionally, most

partitions are generated manually according to the tester's intuition and experiences. Some automation is possible but limited to the syntax information such as data type analysis specific to a programming language. SOA introduces the standard service interface definition and standard data type definitions based on the XML schema. The OWL-S further enables the ontology-based semantic specification of the service IOPE. Such semantic information can improve the ability of machine understandable data definition, and automated generation of the data partitions. For example, if the input parameter of Travelsearch is specified with simple data type string, it can infer the test data of different lengths of strings only. But with the *Accommodation* ontology, it enables searches criteria such as ratings, locations, and prices.

Data pools defined in the TOM are created for each parameter in the service interface specification in OWL-S. Based on the parameter ontology definition, data partitions are generated based on the analysis of parameter class relationships, properties, and restrictions. The restrictions and relationships of the test data partitions can also be derived and verified. In summary, the test case generation process is as follows:

Step 1. Analyze the parameters of the service specified with OWL-S.

Step 2. Create the data pools for the parameters for different test purposes.

Step 3. Derive the class hierarchy for the data partitions for each pool of each parameter.

Step 4. Derive and define the class restrictions and relationships of the data partitions.

Step 5. Derive the data values in each partition.

4.1 Class Hierarchy of Data Partitions

It is an iterative process to derive the test data partition ontology (O^T) in the test domain (T) based on the service ontology (O^S) in the service domain (S). Data partitions are identified hierarchically by recursively exercising the following activities:

1. Direct mapping of the ontology classes in S to T. That is, for each class in S, there is an equivalent class in T. Similarly, The sub-class relationship is also mapped directly from S to T.
2. Class property analysis of S. The test ontology class of the data partition is further partitioned into subclasses with each class representing a data partition to a property of the service ontology.
3. Class relationship and property restriction analysis to remove the redundant classes identified above.
4. Identify class relationship and property restriction of test data partitions.

Taking the *Accommodation* as example, Figure 5 shows the classes hierarchy of the derived TravelTest ontology, which represents the data partitions for testing input parameter *Accommodation*. It first identified six test ontology classes by directly mapping the Travel ontology to the TOM as marked in dark grey. Also the subclass relationships in the Travel ontology were mapped directly to the TravelTest ontology. As *Accommodation* has a property *hasRating* with three values, 18 sub-partitions are further generated by decomposing all the 6 classes with the *hasRating* property. The restrictions on the Travel ontology remove 9 of the derived 18 subclasses.
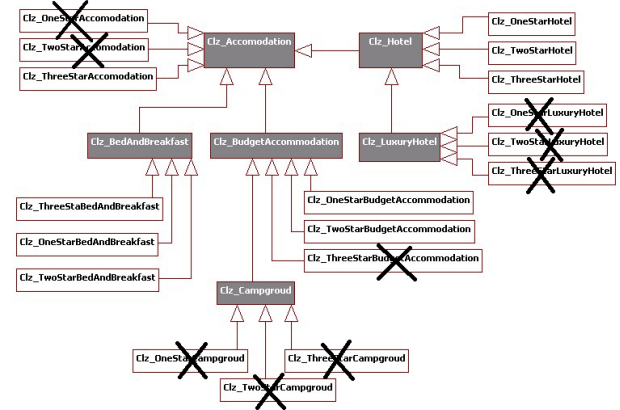

Figure 5 The derived TravelTest ontology

For example, the partitions *Clz_Hotel* and *Clz_LuxuryHotel* are generated as equivalent to the service ontology *Hotel* and *LuxuryHotel* respectively. The subclass relationship between *Hotel* and *LuxuryHotel* is also mapped directly to *Clz_Hotel* and *Clz_LuxuryHotel*. Three more subclasses are further identified for *Clz_ LuxuryHotel* by decomposing it into *Clz_OneStarLuxyHotel*, *Clz_TwoStarLuxyHotel*, and *Clz_ThreeStarLuxyHotel*. However, according to the third restriction, luxury hotel can only be three-star. From this restriction, one can infer that:

1). *Clz_OneStarLuxyHotel*, *Clz_ TwoStarLuxyHotel* do not make sense; and

2). *Clz_ LuxuryHotel* ≡ *Clz_ThreeStarLuxyHotel*

Therefore, the three subclasses are removed due to conflict (1) and duplication (2). Similarly, the sub-partitions {*Clz_OneStar, Clz_TwoStar, Clz_ThreeStar BuddgetAccommodation*} are removed from restriction 1 and the sub-partitions {*Clz_OneStarCampground, Clz_TwoStarCampground, Clz_ThreeStarCampground*} are removed from restriction 2.

4.2 Data Partition Relationship Analysis

OWL supports the following relationship definition among ontology classes:

1. Subclass: It means that if X is a subclass of Y.
2. Equivalent class: Two classes have precisely the same instances and can be replaced alternatively.

3. Disjoint class: A a member of one class cannot simultaneously be an instance of a disjoint class. It indicates that two partitions are non-overlapping.
4. Intersection class: A complex class can be defined by the intersection of classes and/or property restrictions. It means the individuals of the class should satisfy all the classes and restrictions.
5. Union Class: A complex class can also be defined by the union of classes and/or property restrictions. It means the individuals of the class should satisfy any the classes and restrictions.
6. Complementary class: It means that if X is a complementary class of Y, then X takes all the individuals that are not in Y.

For derived data partitions, all the relationships in the service domain can map directly to the test domain. Following is the disjoint relationship on the data partitions that are derived from the example ontology. In this example, *Hotel* and *BedAndBreakfast* are two disjoint classes in the Travel ontology. *Clz_Hotel* and *Clz_BedAndBreakfast* are two corresponding test data partitions which are equivalent to them respectively. Hence, it infers that *Clz_Hotel* and *Clz_BedAnd Breakfast* are also two disjoint classes.

$$(Hotel \cap BedAndBreakfast = \emptyset) \cap$$
$$(Clz\_Hotel \equiv Hotel) \cap$$
$$(Clz\_BedAndBreakfast \equiv BedAndBreakfast) \cap$$
$$(Clz\_TwoStarHotel \sqsubset Clz\_Hotel) \cap$$
$$(Clz\_TwoStarBedAndBreakfast \sqsubset Clz\_BedAndBreakfast)$$
$$\Rightarrow (Clz\_Hotel \cap Clz\_BedAndBreakfast = \emptyset) \cap$$
$$(Clz\_TwoStartHotel \cap Clz\_TwoStartBedAndBreakfast = \emptyset)$$

4.3 Data Partition Property Restriction Analysis

OWL supports the definition of ontology class properties restrictions, including the cardinality constraints and value constraints. The cardinality constraints define the minimum (*min*) and maximum (*max*) number of values of a class property. For example, an accommodation can have exactly one rating. The value constraints define the value scope and range of a property. *owl:allValueFrom* requires that for every instance of the class that has instances of the specified property, the values of the property are all members of the class indicated by the *owl:allValuesFrom* clause. For example, suppose that people can only go hiking (*Activity*) in the national park (*Destination*). Hence, the *isPossibleIn* property of *Hiking* ontology has the restriction of "only *NationalPark*". *owl:hasValue* restriction (*has*) allows us to specify classes based on the existence of particular property values. For example, suppose only 3 ratings are defined for an accommodation, hence the *hasRating* property of *Accommodation* has an *owl:hasValue* constraint as an enumeration of the accepted values "{*OneStar, TwoStar, ThreeStar*}".

The property restrictions of the derived data partitions can also be obtained by mapping directly from the ontology. For example, all the partitions in the class hierarchy have exactly one value of the *hasRating* property. However, as some partitions are derived by property decomposition, the value constraints defined in the ontology usually cannot be mapped directly to the data partitions. For example, the *hasRating* property of *Clz_ThreeStar* can only have one value that is "*ThreeStar*".

4.4 Data Value Generation

The data partitions defined the test classes. Test data instances are generated by filling the class properties with real values. The data values can be generated based on the constraint analysis of the property, especially the value constraint and cardinality constraint. Suppose a property has the cardinality constraint as an interval $[n_1, n_2]$, and the value constraint as an enumeration set $S = \{d_1, d_2, \ldots d_m\}$. To generate the test data values, one can randomly select k data from S where $n_1 <= K <= n_2$.

A database can also be established to import pre-defined data or accumulated historical data. For example, by intercepting the SOAP messages, one can capture the service usage profile and then log the input/output data for future testing. The test generator lookups the database to pick up the values and fills into the test cases.

One can also define rules for deriving data values. Rule language such as SWRL (Semantic Web Rule Language) can support the specification of the dependencies and restrictions of the input data. The following is an example of the companion relationship between two tourists. Two tourists x and y are considered companion if they have the same guide g.

$$def\text{-}has\_Companion = Tourist(?x) \wedge Tourist(?y)$$
$$\wedge\ Guide(?g) \wedge hasGuide(?x, ?g) \wedge hasGuide(?y, ?g)$$
$$\wedge\ differentFrom(?x, ?y)$$
$$\rightarrow has\_Companion(?x, ?y)$$

Suppose that a service requires two inputs parameters of two companion tourists. By reasoning the rule, the test generator can generate the pair of test inputs.

4.5 Ontology-Based C&C Checking

C&C checking is critical to ensure that all the testing assets and service ontologies are properly covered and that no inconsistent or conflicting definitions of the assets can occur. C&C rules are defined to validate the derived data partitions and data values. In general, four categories of rules are defined for classes, relationships,

properties and restrictions respectively. C&C checking are performed from two perspectives:

1. The internal C&C checks the testing assets within the TOM.
2. The external C&C checks the testing assets against the OWL-S specification.

For example, class external completeness requires that in TOM, for each parameter in OWL-S specified services, 1) there is at least one data pool defined; 2) there is at least one corresponding equivalent class defined in the data pool; and 3) each property of the parameter has at least one corresponding data partition. Class internal completeness requires that in the TOM, each data pool has at least one data partition and each data partition has at least one data value.

Consistency checking is based on ontology class computation and reasoning techniques. For example, suppose that 1) o_1, o_2 are two ontology classes in the service domain; and 2) o_1 has a property p, that has the relationship $o_2 \bigcap (o_1 \bigcap hasProperty(p)) = \phi$. Then, 1) two corresponding test classes to_1 and to_2 exist in the test domain, that is, $to_1 \equiv o_1$ and $to_2 \equiv o_2$; 2) to_1 has a sub-class to_{1p} representing a data partition of the property p. That is, $to_{1p} \equiv (o_1 \bigcap hasProperty(p))$; and 3) $to_{1p1} \bigcap to_2 \equiv \phi$.

5 Tool Implementation and Experiments

A prototype tool has been implemented as a RCP (rich client platform) application on the Eclipse platform, as shown in Figure 6. It incorporated with open source projects for OWL editing (Protégé), parsing (Jena2), interpreting (OWL-API) and reasoning (Pellet). The OWL-S analyzer imports and interprets the OWL-S specification of the SUT. IOPE information is collected for test generation. The TOM generator interprets and reasons over the service ontology definition, and generates the test cases according to the TOM. C&C checker validates the derived test cases based on the C&C rules defined offline. A rule engine, Jess, is also integrated to enable rule-based reasoning. The SWRL specified rules were translated into Jess inputs for interpretation.

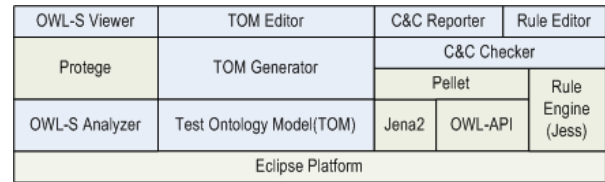| OWL-S Viewer	TOM Editor	C&C Reporter	Rule Editor
Protege	TOM Generator	C&C Checker	
		Pellet	Rule Engine (Jess)
OWL-S Analyzer	Test Ontology Model(TOM)	Jena2	OWL-API	
Eclipse Platform			

Figure 6 Architecture of the Tool Implementation

Figure 7 shows the TOM editor's user interface which recognizes the parameters and ontology classes defined in the service. The screenshot shows the data pool and data partitions generated for Accommodation, and the details of a data partition, including the data values and properties.
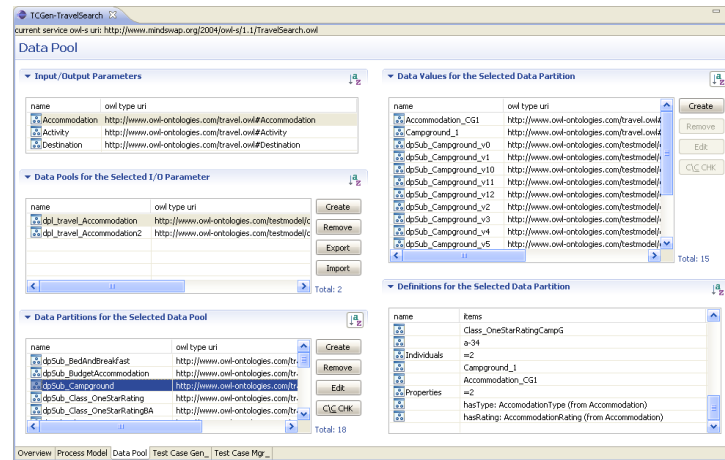

Figure 7 A Screenshot of the tool

Experiments are exercised on a TravelSearch composite service. Ontology are defined for a simplified Travel domain and the OWL-S are specified for the process of TravelSearch which is a composition of three search services including Accommodation Search, FlightSearch and RestaurantSearch. Table 1 shows the partition and data value generated for testing

the FlightInfo input ontology. FlightInfo has four properties including Airline, City, Seat and Time. Based on the property definition, altogether 17 partitions are generated. 7 of the 17 partitions are generated considering the composition of different properties, such as the City and Time. 1413 test data are generated for the 17 partitions. With restriction

analysis, the number of test data is greatly reduced to 344 to remove redundancies and conflictions.

Table 1 Test partition and test data generated in the experiment

Property	Partition	Num of Data (generated)	Num of Data (reduced)
Airline	P1: FlightInfo_Airline1	83	7
	P2: FlightOf_Airline2	91	9
	P3: FlightOf_Airline3	82	7
	P4: FlightOf_Airlinex	86	4
City	P5: FlightInfo_FCity	49	3
	P6: FlightInfo_TCity	49	3
Seat	P7: FlightInfo_FSeat	125	18
	P8: FlightInfo_Bseat	131	17
	P9: FlightInfo_Eseat	121	45
Time	P10: FlightInfo_Time	202	35
Composite	P11: FlightInfo_CaT	64	64
	P12: FlightInfo_CaFS	38	14
	P13: FlightInfo_CaBS	39	9
	P14: FlightInfo_CaES	36	6
	P15: FlightInfo_TaFS	77	26
	P16: FlightInfo_TaBS	72	27
	P17: FlightInfo_TaES	68	50
Total		1413	344

The 344 test data are exercised in two ways. (1) randomly select a number of test cases; and (2) choose a group of partitions and randomly select data from each partitions. The experiments are exercised independently for 10 times. Figure 8 compares the average results of the two ways. It shows that to achieve the same coverage, the partition testing needs much less test cases. For example, to cover 1,550 line of code, the partition-based approach requires about 20 test cases while the random testing requires 60 test cases, as much as three times.

The comparisons between partition testing and random testing have been discussed[14]. In general, partition testing with properly designed partition can achieve better performance than random testing can. This paper shows that the ontology-based approach can improve the adequacy of data partitions and improve the testing effectiveness.
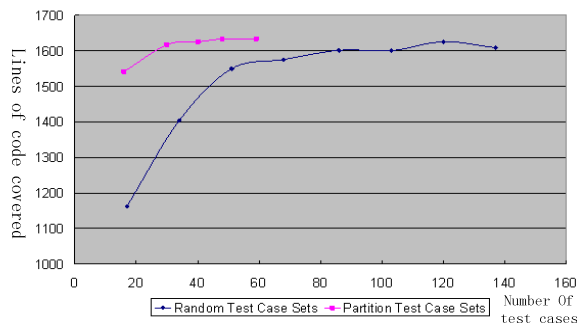

Figure 8 Comparison with random testing

6   Related Works

6.1   Web Services Testing

Recently, testing SOA software receives significant attention. De [9] emphasized that the success of WS depends upon the capability to resolve the test issues. Bloomberg [4] analyzed the three phases of testing technologies development and pointed out that in phase three (2004 and beyond), WS testing should be focused on the features such as dynamic runtime capabilities, WS orchestration testing and WS versioning. Canfora and Penta [6] also discussed the opportunities and challenges in SOA testing from different perspectives, including service developer, provider, integrator, third parties, and end users.

Specification-based testing (or model-based testing) receives significant attention in testing SOA software. Techniques for automatic test case generation and selection are discussed in [2][13][16][17][19]. For examples, Tsai, et al., [17] applied the Swiss Cheese test case generation technique from the Boolean expressions extracted from the conditions and decisions in the WS specifications. Smythe [16] investigated the WS interoperability testing based on a new SoA-Profile of UML. Another common approach is to apply model checking techniques to service specification or service code [9][12].

Testing contracts were also addressed [1][5][11][18]. In [11], Heckel and Lohmann analyzed the three levels of WS contracts representation: the implementation-level, the XML-level, and the model-level. Bruno [5] suggested taking the test cases as contracts to ensure the QoS of services by regression testing with published and agreed test cases. To achieve this, each service will be associated with a XML specification of test suites and QoS assertions. This paper uses ontology information to generate partitions, perform C&C checking, and then generate test inputs from the obtained partitions.

6.2   Partition Testing

Partition testing has been widely used for many years. In [16], Ostrand and Balcer proposed the category-partition method for specification-based functional testing. Categories represented the major properties and characteristics of parameter or environmental conditions, which were derived from the specification for coverage purposes. Each category was partitioned into distinct choices which represented the possible values to test the software. Various methods and techniques have been proposed to achieve cost-effective coverage with error-exposing test cases selected from partitioned sub-domains [7][8]. For example, Chen applied the categories of partitioning to

evaluate the profile-sensitive reliability of software under partition testing [8]. Partition testing can be combined with random testing for complex applications where numerous partitions can be generated [19].

7   Conclusion and Future Work

Using ontology information to generate input partitions and test cases to test SOA software is a promising technique as ontology often provides semantic information not available in service specifications. This paper illustrates this approach and compare with this approach with completely random testing. It shows that this approach can reduce the number of test cases as compare to random testing.

8   Acknowledgements

References

[1]   X. Bai, Y. Wang, G. Dai, W.T. Tsai and Y. Chen, "A Framework of Contract-Based Collaborative Verification and Validation of Web Services". *LNCS 4608*, 2007, pp. 256-271.

[2]   X. Bai, W. Dong, W.T. Tsai, and Y. Chen, "WSDL-Based Automatic Test Case Generation for Web Services Testing," in the *Proceeding of IEEE Int'l Workshop on Service-Oriented System Engineering (SOSE)*, 2005, pp. 207-212.

[3]   X. Bai, G. Dai, D. Xu and W. T. Tsai, "A Multi-Agent Based Framework for Collaborative Testing on Web Services", in the *Proceeding of IEEE Workshop on Collaborative Computing, Integration, and Assurance (WCCIA)*, 2006，pp. 205-210.

[4]   J. Bloomberg (2002), "Web Services Testing: Beyond SOAP", ZapThink LLC, at http://www.zapthink.com.

[5]   M. Bruno, G. Canfora, M. D. Penta, G. Esposito and V. Mazza, "Using Test Cases as Contract to Ensure Service Compliance Across Releases", In the *Proceeding of 3rd International Conference In Service-Oriented Computing(ICSOC'05)*, 2005, pp. 87-100.

[6]   G. Canfora and M. Di Penta. "Testing services and service-centric systems, Challenges and opportunities". *IT Professional*, Vol. 8, No. 2, 2006, pp. 10-17.

[7]   T.Y. Chen, P. Poon, and T.H. Tse. "A Choice Relation Framework for Supporting Category-Partition Test Case Generation", in *IEEE Transactions on Software Engineering*, Vol. 29, No. 7, 2003, pp. 577-593.

[8]   Y. Chen, "Modeling Software Operational Reliability under Partition Testing," In the *Proceeding of IEEE 28th Annual International Symposium on Fault-Tolerant Computing (FTCS-28)*, 1998, pp.314 - 323.

[9]   G. Dai, X. Bai, and C. Zhao. "A Framework for Model Checking Web Service Compositions Based on BPEL4WS". In the *Proceeding of IEEE International Conference on e-Business Engineering*, 2007.

[10] B. De, "Web Services - Challenges and Solutions", WIPRO white paper, 2003, http://www.wipro.com.

[11] R. Heckel and M. Lohmann. "Towards Contract-based Testing of Web Services". *Electr. Notes Theor. Comput. Sci.*, 2005, pp.116:145–156.

[12] H. Huang, W.-T Tsai, R. Paul, "Automated Model Checking and Testing for Composite Web Services". In the *Proceeding of Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2005, pp. 300-307.

[13] S. C. Lee and J. Offutt, "Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis", In the *Proceeding of 12th International Symposium on Software Reliability engineering*, 2001, pp. 200-209.

[14] P. Loo, W. T. Tsai, and W. K. Tsai, "Random Testing Revisited", Information and Software Technology, Vol. 30, 1988, pp. 402-417.

[15] T.J. Ostrand and M.J. Balcer. "The Category-Partition Method for Specifying and Generating Functional Tests". *Communications of the ACM*, Vol. 31. No. 6, 1988, pp. 676-686.

[16] C. Smythe, "Initial Investigations into Interoperability Testing of Web Services from their Specification using the Unified Modeling Language," in *Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, 2006, pp. 95-119.

[17] W.T. Tsai, X. Wei, Y. Chen, R. Paul, B. Xiao, "Swiss Cheese Test Case Generation for Web Service Testing", *IEICE Transactions on Information and Systems*, Vol. E88-D, No. 12, 2005, pp. 2691 – 2698.

[18] W. T. Tsai, Z. Cao, Y. Chen, R. Paul, "Web Services-based Collaborative and Cooperative Computing", *WCCIA 2005, 7th International Symposium on Autonomous Decentralized Systems*, 2005, pp. 552-556.

[19] W. T. Tsai, Y. Tu, W. Shao, and E. Ebner, "Testing Extensible Design Patterns in Object-Oriented Frameworks through Hierarchical Scenario Templates", Proc. of IEEE COMPSAC, 1999, pp. 166-171.

[20] Y. Wang, X. Bai, J. Li, R. Huang, "Ontology-Based Test Case Generation for Testing Web Services," *8th International Symposium Autonomous Decentralized Systems*, 2007, pp. 43-50.

[21] OWL Guide, *http://www.w3.org/TR/owl-guide/*.

[22] OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S.

[23] UML Testing Profile, V1.0, at http://www.omg.org /technology/documents/formal/test_profile.htm.