Test Case Generation of Composite Web Services Based on Semantic Matching and Condition Recognition

Haibo Zhang[1], Hui Xia[1], Xintang Lin[1], Liangjiang Yu[1], Xiangyan Fang[1], Xuan Chen[1], and Hongquan Zhu[2](✉)

[1] Wuhan Digital Engineering Institute, Wuhan 430074, China
work_lxt@126.com

[2] Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China
zhq98@foxmail.com

Abstract. With the rapid development of Internet technology, the application of Web service and their combinations is spreading widely as a vital role. However, due to the black box feature of Web services, test cases of Web service can be obtained only from the perspective of users, which is more challenging than traditional tests. We propose a method based on semantic matching and condition recognition, combined with document parsing to generate test cases of composite Web services. In this method, the first step is to parse the relevant XML documents, then to obtain the parameter types, keywords, as well as conditions and orders in the control flow. The following step is to match the parameter instances according to the conditions in the process or the semantic knowledge base. The last is to encapsulate them into test cases. Experimental results show that our method can generate test cases with low redundancy and high coverage, which can cover more than 85% nodes and paths in BPEL.

Keywords: Composite Web services · Test case generation · Document parsing · Semantic matching

1 Introduction

Web services [1] adopt a Service-Oriented Architecture (SOA) [2] to achieve service invocation through the interaction among entities such as service providers, registration centers and service requesters, without depending on language, platform and protocol. With the rapid development of Internet, cloud computing technology and service-oriented technology tend to mature, Web services have also been put in wide utilization. Developers combine several atomic Web services in accordance with a certain process to provide users with more comprehensive value-added services. It is necessary to test them fully to guarantee the accuracy and stability of the combined Web services.

The generation [3] of test cases is the first stage of composite Web service testing. This paper aims to achieve automated generation of test cases at a relative low cost and high coverage. In this paper, we propose a method to generate test cases for composite

Web services based on semantic matching and condition recognition, and verify the effectiveness of this method through comparative experiments.

The rest of this paper is as follows. Related work is discussed in Sect. 2. Parsing XML documents related to Web services is in Sect. 3. How to generate test cases based on semantic matching and condition recognition is presented in Sect. 4. Experiments and evaluation are in Sect. 5. The last part is conclusion and future work.

2 Related Work

Test case generation is crucial for software testing, model-based test case generation is the most popular method [1, 4–8]. This method improves the accuracy of test cases, but few of them are fully automated without the testing of atomic Web services.

Zhou et al. [9] proposed a test case generation method based on parsing document and solving constraint, which parses XML documents related to composite Web Services. They obtain and encode constraint conditions, and then use the Z3-Solver [10] to solve the encoded constraints, and finally combine the SOAP protocol to generate test cases. However, the speed that Z3 processes string type variable is relatively slow.

Estero-Botaro et al. [11] proposed a genetic algorithm with some bacterial algorithm characteristics to generate a test suite for mutation testing. Experiment shows that the error detection rate and coverage of test cases rise greatly, but costs a lot of time.

In addition, Sun C et al. [12] and Mei et al. [13] studied BPEL runtime testing from the perspective of workflow, and proposed a scenario oriented testing framework for the runtime binding characteristic of composite Web services.

3 Document Parsing

Since service integrators cannot obtain the source code of Web service, test cases can only be generated with the information from interface documents [14]. This section will introduce the parsing method of XML documents in composite Web services in detail.

3.1 XML Documents in Web Services

XML documents related to Web services include BPEL, WSDL and XSD documents, which are respectively used to describe the business processes, interface information, and variable format information. The relationship is shown in Fig. 1 [9].

The basic unit of BPEL [15] is activity, which can be divided into basic activities, structured activities and fault handling activities. In addition to the above three types of activities, there are some nodes in the BPEL documents that describe the external service addresses, partner links, and variable declarations of Web service references.

WSDL document contains nodes: <service>, <binding>, <portType>, <message> and <types>, which are respectively used to describe the information of interface address, communication protocol, operation, message and variable type.

XSD document makes formal definition of the variables with XML format, whose composition structure is in the XSD part in Fig. 1.

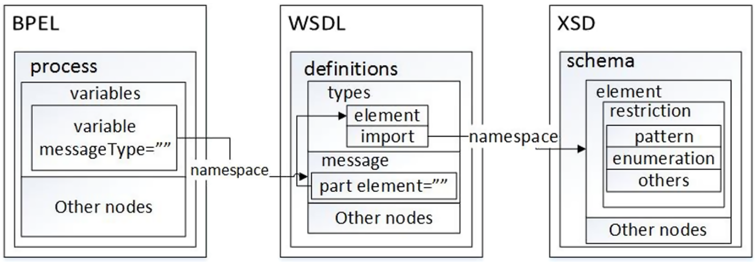


Fig. 1. Relationship of documents related to Web services

3.2 Analytical Method

When parsing the composite Web service document, we start with the BPEL document, where all the atomic Web services and business processes are defined. To construct the controlling flow graph, it starts from the root node <process> to all the nodes in turn, and stores the relationship between nodes and information of themselves. We obtain node information such as loop conditions and branch conditions, and then obtain child nodes iteratively.

Then, we parse WSDL documents. We first retrieve all the <service> elements, for which we jump to <binding> and the elements that match the service. Thus, we get a set of operations (<operation>), and then jump to <portType> to find the input and output <message> corresponding to each operation. Next, we parse the <message> and <types> nodes to obtain the constraint information. Further, it needs to obtain the namespace of the WSDL file for parsing the nodes and constructing test cases.

When parsing XSD documents, we still need to traverse all the nodes to obtain constraint conditions, such as the ordering relationship, values, types, lengths, value ranges, and regular expressions of variables. In addition, the variable definitions in the WSDL file need to parse the XSD file with the same namespace.

4 Test Case Generation Based on Semantic Matching and Condition Recognition

Former test cases generated based on interface parameter types have high redundancy, mainly because many invalid cases are generated based on the type information instead of taking the actual meaning of the parameters into account. For example, in the Web service of attribution query of domestic mobile phone numbers, when we input parameter named “mobileCode” with type “string”, it is likely to generate a bunch of meaningless strings if we only judge by the type “string”.

Likewise, we also need to generate test cases in accordance with various process conditions. For example, a loan service needs to deal with business processes according to the user’s loan amount.

In this section, we extract the input parameter keywords and process conditions by parsing the XML documents, and generate test cases on the basis of semantic matching and condition recognition.

4.1 Keyword Extraction Optimization

Before the semantic matching, we must preprocess the parameter names and operation names to extract keywords. Since different developers have different definitions, the parameter names are irregular and require text processing to find the keywords that can explain the parameters most clearly. We propose a method to deal with the parameter names by analyzing the thousand parameter names in WSDL documents:

(1) Consider IO (input/output): Many inputs and outputs of the same service share the same forms, such as input in the form of "getAbyB". Obviously, B is the input keyword and A is the output keyword. For example, we can extract the keyword "Currency" in the service "currencyService" with an operation "getCurrencyRate".
(2) Separate Words: Considering the definition of parameter names without spaces, we separate these words according to the capitalized first letter.
(3) Filter meaningless words: After separation, some high-frequency but meaningless words should be ignored. The frequency of occurrences of such words in WSDL exceeds 90%, which are saved in a table of our database.

4.2 Building a Local Semantic Library

Building a local semantic library is to facilitate the semantic matching method. We combine both the automatic and manual method to construct the local semantic library:

(1) Automatic method: Present Web services sometimes provide interfaces for the access of specific values allowed by them. For example, a weather forecast service [16] offers weather forecast of 400 cities through the interface "getWeatherby-CityName". We call these interfaces for the legal input parameter value, tag them (the name of a city) with labels like weather, city, and then store into the database.
(2) Manual method: For various parameter types, we have designed a table with artificial maintenance. For example, for the type "double", 3.2, −983.2 and types like "dateTime", "int" etc. are stored in the database. With this table, the program only needs to parse the parameter type for the corresponding input instance.

4.3 Parameter Matching Based on the Semantic Library and DBpedia Domain Knowledge

As the example of currencyService (Sect. 4.1), an operation named "getCurrencyRate" queries the exchange rate between two currencies by inputting two currency codes with an output named "getCurrencyRateReturn". Through keyword extraction, the keyword is "Currency". Then we utilize the local library and DBpedia to match the keywords in the service elements. The parameter matching mechanism is as follows:

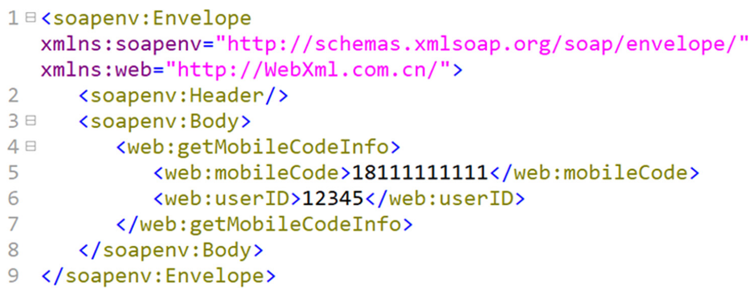
(1) Query the local semantic library according to the input parameter type for the suitable preset input instance, which should be listed to the instance table.
(2) Add the matching record to the instance table if it is in the local semantic library, according to the "keyword" after parameter optimization.
(3) If there's no instance to match, we call the DBpediaSpotlight service. We filter parameters by configuring and input the keyword and its description. Spotlight will respond content, which contains the DBpediaURI that matches the keyword, then extracts the string that can be used as an input parameter and adds it to the instance table.

4.4 Conditional Recognition

As mentioned in the previous section, only a part of the input values is obtained according to semantic matching, while others involve in controlling the BPEL process. Thus, we also need to consider the conditions (<condition>). For example, a parameter "amount" of a loan approval service requests users to input the loan amount. The program will estimate the loan risk based on the amount. For low-risk customers whose request is less than 10,000, the loan will be automatically approved. If it is more than 10,000, the approver will check the loan with an <if> node in the process. Based on the analysis result of the BPEL documents, we will identify the corresponding node for condition judgment and match it with the input parameters.

4.5 Test Case Packaging

SOAP, as a protocol specification for exchanging data, is often used to exchange structured and fixed information in a distributed environment. We need to encapsulate the test data in a SOAP message to generate test cases. A test case for mobile phone number location query service [17] is shown in Fig. 2.



```
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:web="http://WebXml.com.cn/">
   <soapenv:Header/>
   <soapenv:Body>
      <web:getMobileCodeInfo>
         <web:mobileCode>18111111111</web:mobileCode>
         <web:userID>12345</web:userID>
      </web:getMobileCodeInfo>
   </soapenv:Body>
</soapenv:Envelope>
```

Fig. 2. Mobile phone number location query service.

5 Experiment and Evaluation

5.1 Experimental Data and Environment

The hardware environment of the experiment is an Intel Core i5-6400 quad-core processor, 8G memory and the operating system is Windows 10. The development environment is jdk1.8 and Eclipse Oxygen.

We selected some typical composite Web services, including Loan Approval Service (LAS), BookLoan, FlowLinks, and CaculatorProj. These services involve various of activities in Sect. 3.1. The scale of the program is shown in Table 1. LOC represents the number of lines of BPEL, and AN (activities number) is the number of activities.

Table 1. Composite service information and experimental results.

Composite Web service	BPEL LOC	AN	Test case evaluation				
			Total	Redundancy	Node coverage	Path coverage	Time (ms)
CombineUrl	38	4	2	0	100%	100%	93
HelloWorld	45	4	2	0	100%	100%	107
While	82	8	2	0	100%	100%	120
Alarm	86	11	3	33%	100%	100%	142
CaculatorProj	151	13	3	0	100%	100%	97
LAS	204	14	7	40%	100%	100%	280
FlowLinks	92	22	4	33%	90%	86%	236
BookLoan	251	37	8	20%	89%	91%	302

5.2 Analysis of Results

In order to evaluate our method, we chose some common evaluation indicators, including coverage, redundancy, generation time, etc. Since the composite Web service was fully tested before the release, and there is no publicly known bug set, we have not considered the evaluation of the ability to search for bug of the test case.

In Table 1, the evaluation indicators of test case are as follows:

(1) Total: the number of all test cases generated in one execution;
(2) Redundancy: the proportion of test cases that overlap with nodes or paths covered by others;
(3) Node coverage: the proportion of nodes covered by the generated test case set;
(4) Path coverage: the proportion of paths covered by the generated test case set;
(5) Time (unit: milliseconds): the average time required to generate a test case.

The experimental results are shown in Table 1. We found that for small-scale composite Web services (the first four in Table 1), we can achieve 100% coverage with fewer test cases but almost no redundancy. For more complex services (the following four in Table 1), the coverage may decrease, but still with little redundancy and low time cost. Compared with other work, the number of test cases generated by our method is less, because generating by semantic matching is more practical, and the scope of parameters is constrained by condition recognition.

5.3 Comparative Experiment

To verify the effectiveness, we set three experiments: (a) remove semantic matching; (b) remove condition recognition; (c) remove both. The results are in Table 2.

Table 2. Comparison of node coverage.

Composite Web service	Result	Control group A	Control group B	Control group C
CombineUrl	100%	100%	100%	100%
HelloWorld	100%	100%	100%	100%
While	100%	50%	50%	0
Alarm	100%	100%	36%	36%
CaculatorProj	100%	100%	38%	38%
LAS	100%	60%	57%	35%
FlowLinks	90%	32%	41%	23%
BookLoan	89%	41%	32%	27%

By comparison, we found that when any one of the core points “semantic matching” and “conditional recognition” is removed, the effect of test case generation will shrink. When both are removed, similar to the traditional test case generation method based on interface parameter types, the coverage of test cases is sharply reduced. Therefore, using semantic matching and conditional recognition techniques is necessary (Table 3).

Table 3. Comparison of path coverage.

Composite Web service	Result	Control group A	Control group B	Control group C
CombineUrl	100%	100%	100%	100%
HelloWorld	100%	100%	100%	100%
While	100%	50%	0	0
Alarm	100%	100%	0	0
CaculatorProj	100%	100%	0	0
LAS	100%	66%	66%	33%
FlowLinks	86%	25%	50%	25%
BookLoan	91%	40%	40%	20%

6 Conclusion and Future Work

In this paper, we introduce a method for test case generation of composite Web service. This method parses XML documents to obtain semantics, constraints, and other information related to input parameters, overcoming the problem of high redundancy of test cases used to be merely based on interface parameter type.

In the future, we plan to incorporate mutation strategies into our test case generation, combined with fuzzing technology to strengthen the detection of service vulnerabilities and to improve coverage.

References

1. Yan, J., Li, Z., Yuan, Y., Sun, W., Zhang, J.: BPEL4WS unit testing: test case generation using a concurrent path analysis approach. In: 2006 17th International Symposium on Software Reliability Engineering, pp. 75–84. IEEE (2006). https://doi.org/10.1109/issre.2006.16
2. Krafzig, D., Banke, K., Slama, D.: Enterprise S.O.A. Service-Oriented Architecture Best Practices. Prentice Hall PTR, Indiana (2004)
3. Wurth, E., Delpiroux, J.: Web service testing (Google Patents, 2018). http://www.freepatentsonline.com/y2016/0127409.html
4. Mei, L., Chan, W., Tse, T.: Data flow testing of service choreography. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 151–160 (2009). https://doi.org/10.1145/1595696.1595720
5. Mei, L., Chan, W., Tse, T.: Data flow testing of service-oriented workflow applications. In: Proceedings of the 30th International Conference on Software Engineering, pp. 371–380 (2008). https://doi.org/10.1145/1368088.1368139
6. Hou, S.-S., Zhang, L., Lan, Q., Mei, H., Sun, J.-S.: Generating effective test sequences for BPEL testing. In: 2009 Ninth International Conference on Quality Software, pp. 331–340. IEEE (2009). https://doi.org/10.1109/qsic.2009.50
7. Wu, C.-S., Huang, C.-H.: The web services composition testing based on extended finite state machine and UML model. In: 2013 Fifth International Conference on Service Science and Innovation, pp. 215–222. IEEE (2013). https://doi.org/10.1109/icssi.2013.46
8. Ni, Y., et al.: Effective message-sequence generation for testing BPEL programs. IEEE Trans. Serv. Comput. **6**, 7–19 (2013). https://doi.org/10.1109/TSC.2011.22
9. Zhou, L., Xu, L., Xu, B., Yang, H.: Generating test cases for composite web services by parsing XML documents and solving constraints. In: 2015 IEEE 39th Annual Computer Software and Applications Conference (2015). https://doi.org/10.1109/compsac.2015.51
10. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 114–124 (2013). https://doi.org/10.1145/2491411.2491456
11. Estero-Botaro, A., García-Domínguez, A., Domínguez-Jiménez, J.J., Palomo-Lozano, F., Medina-Bulo, I.: A framework for genetic test-case generation for WS-BPEL compositions. In: Merayo, M.G., de Oca, E.M. (eds.) ICTSS 2014. LNCS, vol. 8763, pp. 1–16. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44857-1_1

12. Sun, C., Zhao, Y., Pan, L., Liu, H., Chen, T.Y.: Automated testing of WS-BPEL service compositions: a scenario-oriented approach. IEEE Trans. Serv. Comput. **11**, 616–629 (2018). https://doi.org/10.1109/TSC.2015.2466572
13. Mei, L., et al.: A subsumption hierarchy of test case prioritization for composite services. IEEE Trans. Serv. Comput. **8**, 658–673 (2015). https://doi.org/10.1109/TSC.2014.2331683
14. Zhao, H., Chen, J., Xu, L.: Semantic web service discovery based on LDA clustering. In: Ni, W., Wang, X., Song, W., Li, Y. (eds.) WISA 2019. LNCS, vol. 11817, pp. 239–250. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30952-7_25
15. Jordan, D., et al.: Web services business process execution language version 2.0. OASIS Stand. **11**, 5 (2007). http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
16. Weather Web Service. http://ws.webxml.com.cn/WebServices/WeatherWebService.asmx
17. MobileCode Web Service. http://ws.webxml.com.cn/WebServices/MobileCodeWS.asmx