CHAPTER - 6

Black-box Testing of Service-Oriented Software

One major flaw of white-box testing approach for web services is that web services are assumed to be transparent, which means that the tester has knowledge about internal structure, source code etc. which is unrealistic in some cases. Therefore, in this chapter we are going to test web services with no knowledge of internal details i.e black-box testing and with partial knowledge of internal details, called gray-box testing. This chapter is organized as follows. Section 6.1 describes the black-box testing of web service using program slicing artifacts. Section 6.2 illustrates two approaches for model-based testing one using BPMN and second using SoaML service interface diagram.

6.1 Black-box Testing of Web Services Through Program Slicing Artifacts

Web services and service-oriented computing (SOC) paradigm have received significant attention due to its widespread adoption and promotion by major IT vendors. As more and more service-oriented softwares are built today, testing of web services has become the crucial point. Web services are distributed over various sites, maintain loosely coupled relationship among them, discoverable through UDDI, hide internal logic (abstraction), minimize retaining information (statelessness) and adhere to service level agreement (SLA). These inherent characteristics impose great challenges to the tester. In this section, we try to imposes a hierarchical structure on web service description language (WSDL) in order to uncover more and more errors through program slicing artifacts like dependence graph. Our work carries out black-box testing based on web service description language (WSDL) and its dependencies induced within. Our approach impose a hierarchical structure on WSDL, intro-
duces data and control dependencies, and introduces web service description graph (WSDG) as semantic element which will be used for testing.

The world wide web consortium (W3C) [238] defines web service as “a software system designed to support interoperability between machine-to-machine interaction over a network”. It has an interface described in a machine-processable format, WSDL. Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically transported using HTTP with an XML serialization in connection with other web-related standards [239]. Web services use standard protocols such as web service description language (WSDL) to describe service descriptions, universal description discovery and integration (UDDI) to publish WSDL, and simple object access protocol (SOAP) to exchange messages among services.

Web services are loosely coupled, which means that each of its components has little knowledge of all the other components. When there are any changes in client interface of the web service, there is no need to change server interface of the web service. In simple words, changes in the interface of web service do not stop the client from interacting with web service. Web services are coarse-grained, which imply that they consist of a small number of large components. Web services can have both synchronous and asynchronous connection between client and service execution. The asynchronous operations, when invoked, do not block the client while service is completing its operation. The client will receive the result from asynchronous operations later than from synchronous operations. Web services having asynchronous capabilities are necessary to have loosely coupled systems.

Web services were designed to support interoperability between different heterogeneous machines. Web services communicate via SOAP messages [240], allow them to communicate with any application, written in any language or designed for any platform, which can understand SOAP. However, like all software, such an application should be tested before it is delivered to the user. Web service is described by a web service description language (WSDL) document [241]. WSDL was established as a standard by the world wide web consortium (W3C) [238]. The WSDL document describes web service in terms of its interfaces with operations, types of data the web service takes or returns. WSDL does not provide a description of the internal operation of web service. Nevertheless, such a WSDL document can be the basis for testing the web services.

A typical web service interaction includes interaction among three standardized software components: service provider, service registry and service consumer as shown in Fig. 6.1. The service provider design WSDL document and publishes in a service registry (UDDI) through technical Model (tModel), which stores references for WSDL document. The service consumer looks up the service registry (UDDI) for a possible match against business
requirements, finds the service endpoint (URL), and finally calls the matched web service by exchanging SOAP messages.

![Diagram of a web service interaction](image)

**FIGURE 6.1: A web service interaction**

The rest of the Section is organized as follows. Section 6.1.1 describes the structure of WSDL document. Section 6.1.2 describes greatest common divisor (GCD) web service example. Section 6.1.3 describes the inherent dependencies of WSDL document. Section 6.1.4 presents black-box testing of web services using extended WSDL. Section 6.1.5 presents our proposed extension, WSDG. Section 6.1.6 describes the experimental results. Section 6.1.7 compares our work with some related work.

### 6.1.1 Structure of WSDL

Testing based on WSDL requires us to understand the structure and semantic of WSDL document. In this section, we have discussed two recommended specification standards WSDL 1.1 and WSDL 2.0 in detail.

The newest version of WSDL is 2.0 which is recommended by W3C, however, WSDL 1.1 is still quite common and many software support this version. We have used WSDL 1.1 for our work. Fig. 6.2 illustrates the major differences between both versions. Sections `<messages>` and `<portType>` were combined to create new section `<interface>`.

WSDL 2.0 is divided into several sections as below:

- **description**: is a container, inside of which the remaining sections are located.
- **types**: describes the data types used by a web service.
• **interface**: describes the abstract functionality the web service provides (what messages it sends and receives, and possible fault messages).

• **binding**: provides information how to access the service, and

• **service**: provides information where to access the service.

There are two optional sections in WSDL document. They are as follows:

• **documentation**: provides human-readable description of a web service, and

• **import**: is used to import other XML schemas.

The detailed description of each section of WSDL 2.0 is given as below:

• **description**: The *description* tag contains all other tags and is used for declaring namespaces. Element *targetNamespace*, contain link to WSDL document describing the service.

• **types**: The *types* tag contains data types that the web service uses. Types, to be declared, have to be supported by web service API. Each operation can have its own input type (request), output type (response) and fault (error). The types defined in *types* can
be a simple type such as string or int. Complex type such as response can consist of many elements, which can be built-in data types.

- **interface**: The *interface* tag is a set of operations representing the interaction between the user and the web service. Each of these operations specifies the sequence of messages exchange that service sends or receives and the type of a given message. Operations are similar to methods or functions in programming languages.

  WSDL defines four types of operations:

  - **One-way**: operation can receive a message.
  - **Request-response**: operation can receive a request and will return a response.
  - **Solicit-response**: operation can send a request and will wait for a response, and
  - **Notification**: operation can send a message.

- **binding**: The *binding* tag is responsible for access to the web service; it describes the protocol, which may be used. This is a way to bind web service with the protocol. If there is defined binding for given interface, then bindings for all operations from that interface must be defined.

- **service**: The *service* tag contains a set of endpoints at which service is provided. The endpoints are places at which the service is provided.

Similarly, WSDL 1.1 is divided into the same sections as defined for WSDL 2.0 specification as given below:

- **definition**: is a container, similar to *description* of WSDL 2.0 specification. It serves as the root or parent element of every WSDL document. It also houses all other parts of the service definition and the location in which the many namespaces used within WSDL documents are established.

- **types**: The data types can be mapped to and from the types of conventional programming languages and is similar to types of WSDL 2.0 specification.

- **portType**: It represent collections of operations, similar to *interface* of WSDL 2.0 specification.

- **port**: It describes an endpoint where operation resides defined by the network address and binding. It contains the location information of web services.
- **binding**: It associates a port type to a protocol and data format. It provides information how to access the service.

- **service**: It is collection of ports. It provides information where to access the service.

WSDL enables loose coupling among web services. It contains an abstract description (portType or interface, operation, and message) and concrete description (binding, port, and service). However, this information is not sufficient to test web services, the tester also requires code to determine possible errors, which is not possible with published web services. Due to unavailability of such code, the tester may opt for black-box testing where tester observes SOAP input and output messages available from WSDL descriptions. However, one should mention that even after satisfactorily carrying out testing it is impossible to guarantee that the software is error-free [194].

### 6.1.2 A Web Service Example: Greatest Common Divisor (GCD)

To understand our proposal, we need to understand how a web service is developed and consumed. This section describes how to create a web service, its related artifacts XSD and WSDL.

Let there is a business requirement to create a web service which computes greatest common divisor of two input values of type integer and returns an integer as the result. To fulfill this requirement, first, we create an XML schema definition (XSD) file. We generated this XSD file using NetBeans IDE [168] plugged with SOA and XML plugins as shown in Fig. 6.3. The XSD file defines two complex types `compute_gcd` and `compute_gcdResponse`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:tns="http://mypackage/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
  targetNamespace="http://mypackage/"/>
<xs:element name="compute_gcd" type="tns:compute_gcd"/>
<xs:element name="compute_gcdResponse" type="tns:compute_gcdResponse"/>
<xs:complexType name="compute_gcd">
  <xs:sequence>
    <xs:element name="arg0" type="xs:int"/>
    <xs:element name="arg1" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="compute_gcdResponse">
  <xs:sequence>
    <xs:element name="return" type="xs:int"/>
  </xs:sequence>
</xs:complexType>
</xs:schema>
```

**FIGURE 6.3:** Generated XSD of `compute_gcd` web service

To understand our proposal, we need to understand how a web service is developed and consumed. This section describes how to create a web service, its related artifacts XSD and WSDL.

Let there is a business requirement to create a web service which computes greatest common divisor of two input values of type integer and returns an integer as the result. To fulfill this requirement, first, we create an XML schema definition (XSD) file. We generated this XSD file using NetBeans IDE [168] plugged with SOA and XML plugins as shown in Fig. 6.3. The XSD file defines two complex types `compute_gcd` and `compute_gcdResponse`. The `compute_gcd` type has two elements `arg0` and `arg1` of `xs:int` type, and the element
compute-gcdResponse has an element return of type xs:int, which are equivalent to integer data type. After that, we have created two elements namely compute-gcd and compute-gcdResponse respectively.

FIGURE 6.4: WSDL of compute-gcd web service

This XML file does not appear to have any style information associated with it. The document tree is shown below:

```xml
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ns="http://example.com"
    targetNamespace="http://example.com"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <xsd:schema
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:tns="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/soap/Envelope/"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      targetNamespace="http://example.com">

    </xsd:schema>
  </types>

  <message name="compute_gcd">
    <part name="parameters" element="tns:compute_gcdRequest"/>
  </message>

  <message name="compute_gcdResponse">
    <part name="parameters" element="tns:compute_gcdResponse"/>
  </message>

  <portType name="compute_gcd">
    <operation name="compute_gcd">
      <input message="tns:compute_gcdRequest"/>
      <output message="tns:compute_gcdResponse"/>
    </operation>
  </portType>

  <binding name="compute_gcdBinding" type="tns:compute_gcd">
    <soap binding transport="http://schemas.xmlsoap.org/soap/http">
      <operation name="compute_gcd">
        <soap:operation soapAction=""/>
      </operation>
    </soap>
  </binding>

  <service name="compute_gcdService">
    <port name="compute_gcdPort" binding="tns:compute_gcdBinding"/>
    <soap:address location="http://localhost:8080/CXCD/CXCDService"/>
  </service>
</definitions>
```
defined earlier as in Fig. 6.3. It includes the XSD file with attribute `schemaLocation` and define elements like `PortType`, `Operation`, `Messages`, `Port`, `Binding` and `Service`. These elements represent service/object, method, argument(s), endpoint, transport protocol and a set of endpoints respectively. These elements have already been discussed in the previous Section too. The numbers assigned on the left side of each element will be used to describe inherent dependencies and have been discussed in next subsection.

Next, we create a web service named `compute_gcd` as shown in Fig. 6.5. The `@WebService()` annotation describes the class `CGCD` as web service. The WSDL file should be published in the service registry (UDDI) since NetBeans IDE does not support UDDI, the service client must acquire it manually. Then the client communicates with web service by exchanging SOAP messages. First, we clean, build and deploy the web service successfully. Next we tested `CGCD` web service using NetBeans tester service by inputting 6 and 12 shown in Fig. 6.6. By clicking the button `computeGcd`, we actually invoke the web service using SOAP request message containing the input parameters and gets the SOAP response as 6 which is shown in Fig. 6.7. Such type of testing is called black-box testing where a tester observes the input and output without knowing the internal code or structure.

```java
package mypackage;

import javax.jws.WebService;

/**
 * @author KKR
 */
@WebService()
public class CGCD {
    public int compute_gcd(int x, int y) {
        while (x != y) {
            if (x > y)
                x = x - y;
            else
                y = y - x;
        }
        return x;
    }
}
```

FIGURE 6.5: `compute_gcd` web service

In order to test a web service before its final deployment, the only thing available to tester is WSDL. Unfortunately, WSDL contains information which is insufficient to test. To test
a web service based on WSDL, a tester needs to know the structure of web service i.e. its control and data dependencies through which tester can check the service for possible errors. In next section, we propose our extension to WSDL document to get web service tested.

**CGCDService Web Service Tester**

This form will allow you to test your web service implementation (WSDL File)

To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.

**Methods:**

```java
public abstract int mypackage.CGCDD.computegcd(int, int)
```

**FIGURE 6.6:** NetBeans tester service to invoke `compute_gcd` web service

**computeGcd Method invocation**

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>6</td>
</tr>
<tr>
<td>int</td>
<td>12</td>
</tr>
</tbody>
</table>

**Method returned**

int : "6"

**SOAP Request**

```xml
<soapenv:Body>
<compute_gcd xmlns="http://mypackage/">
<arg1>6</arg1>
<arg2>12</arg2>
</compute_gcd>
</soapenv:Body>
</soapenv:Envelope>
```

**SOAP Response**

```xml
<soapenv:Body>
<compute_gcdResponse xmlns="http://mypackage/">
<return>6</return>
</compute_gcdResponse>
</soapenv:Body>
</soapenv:Envelope>
```

**FIGURE 6.7:** SOAP response for `compute_gcd` web service
6.1.3 Inherent Dependencies of WSDL

The WSDL possesses inherent dependencies due to its various predefined semantic elements. These dependencies can be statically analyzed and can be represented as the graph in Fig. 6.8. It shows that when a Service Consumer invokes the service through WSDL, the service element is first invoked. The service element further invokes port and its attribute binding. The port element invokes portType, operation and message elements recursively. And finally, service gets executed on these ports along with associated operations with input/output messages.

These dependencies do not help much in testing as these dependencies are static and never change during the whole lifetime of service unless the service gets maintenance or become obsolete.
6.1.4 Black-box Testing of Web Service Using Extended WSDL

Our proposed approach for black-box testing of web services using extended WSDL can be visualized as shown in Fig. 6.9. We present our proposed approach as an algorithmic pseudo-code as shown below in Algorithm 1:

**Algorithm 1** Black-box Testing of Web Service Using Extended WSDL

1. Create extended XSD file and import it in WSDL through `schemaLocation` attribute.
2. Validate extended WSDL and XSD using NetBeans IDE.
3. Create web service.
5. Finally, run the web service client, generate test cases, run the test cases and check the results.

**FIGURE 6.9:** Our proposed approach for black-box testing using extended WSDL.
6.1.5 Extensible Element Web Service Description Graph \texttt{<WSDG>}:
Our Proposed Extension to WSDL Using XSD

An XML schema (XSD) is used to provide the syntactic rules for XML data and documents that must be obeyed. The XSD documents describe the elements and their attributes just like the type of content that the elements can have, detail the order in which elements can appear or provide a choice of elements for a given context. One can think XSD as the metadata for the XML document. Similarly, DTD also provides the basic structure of XML documents (limited to elements and attributes). An XSD is a newer format and can do everything a DTD can, along with additional restriction and facets. The other advantage of XSD schema over DTDs is that they are XML based, because of that they are easier to parse and display in a hierarchical manner. Because of these advantages, we prefer to use XSD over DTD for our proposed extension of WSDL.

A web service may contain a number of classes, and these classes may, in turn, contain a number of methods, more precisely web methods. These web methods may contain data and/or control dependence statements. The control or data dependence statements may be of sequence, selection and iteration constructs [194]. This observation can be made as a set of rules which can be expressed using an XSD. We describe these rules by defining an extensible element \textit{Web Service Description Graph} \texttt{<WSDG>} which serves as root and others like classes, methods, data dependencies and control dependencies will become subchild of it. We explicitly describe these constructs as semantic elements in our extensions. The extended XSD schema is shown in Fig. 6.10. It should be noted that the XSD schema is a tree containing a single root along with leaf and non-leaf nodes.

The schema code has been shown in Fig. 6.11. It defines the WSDG which is essential for testing services as global elements of complex type wsdg in modified XSD document. The wsdg contains element class of complex type \textit{Class} and an attribute \textit{Version} of type integer. The class encloses element method of complex type \textit{Method} and attributes \textit{Cname}, \textit{CVisibility} of type string and visibility respectively. The method encloses two elements cdg and ddg of complex types \textit{CDG} and \textit{DDG} in sequence respectively. It also contains attributes like \textit{Mname}, \textit{Argument} and \textit{Visibility} of types string, argument and visibility respectively. The cdg covers three elements \textit{Seq}, \textit{Selection}, \textit{Iteration} of complex types \textit{seq}, \textit{selection} and \textit{iteration} in a sequence respectively. Similarly, the ddg covers three elements \textit{seq}, \textit{selection} and \textit{iteration} in a sequence respectively. The seq, selection and iteration covers element \textit{statement} of type integer. In short, this XSD file imposes hierarchical structure on WSDL through XSD and sets structural information for testing.
FIGURE 6.10: Extended XSD design
FIGURE 6.11: Extended XSD for compute_gcd web service
6.1.6 Implementation and Experimental Results

In the followings, we discuss the black-box testing of extended XSD, black-box testing of extended WSDL, black-box testing of our compute gcd web service, an important feature of our proposed extention, WSDL data sets, and experimental results.

6.1.6.1 Black-box Testing of Extended XSD

We black-box tested our extended XSD using NetBeans IDE. To test our proposed extention (Fig. 6.11), we have created an XML file shown in Fig. 6.12. The XML file uses xsi:noNamespaceSchemaLocation attribute to refer the location of extended schema. For black-box testing, we used equivalence class partitioning method. Since the input domain of our compute gcd web service is a range of values, we devise three test cases; one valid and two invalids. The valid test case is: $\langle \text{statement} \rangle 1 \langle /\text{statement} \rangle$ and invalid test cases are $\langle \text{statement} \rangle \langle \text{statement} \rangle, \langle \text{statement} \rangle ABC \langle /\text{statement} \rangle$. Next, we tested the schema using NetBeans validation feature. The result of valid test execution is shown in Fig. 6.12 and invalid test case execution result is shown in Fig. 6.13. Testing any representative value from an equivalence class is similar to test for all the values from it. So, we black-box tested the web services using representative test cases which demonstrate the applicability of our proposed extension.
6.1.6.2  Black-box Testing of Extended WSDL

Next, we tested the extended WSDL with NetBeans IDE. We used WSDL editor to design schema as shown in Fig. 6.14. We included extended XSD file in WSDL using schemaLocation attribute to point to its location/URI as shown in Fig. 6.15. The rounded rectangle portion highlights our extension to WSDL through XSD. It defines root element WSDG and sub-child’s cdg etc. in the form of loops (Iteration), choices (Selection). Every service provider must accommodate its XSD and WSDL as shown in Fig. 6.11 and in Fig. 6.15 accordingly and provide the required structural information. The result of executing valid test cases on this WSDL is shown in Fig. 6.16. Now onwards this WSDL will serve as standard template or contract and service provider must follow it to get its service to be testable.
FIGURE 6.15: Extended WSDL for compute_gcd web service
6.1.6.3 Black-box Testing of compute\_gcd Web Service

At last, we tested compute\_gcd web service by creating service consumer program as shown in Fig. 6.17. We found that the extension that we proposed are correct with respect to the extenstions that we applied in XSD and corresponding WSDL files.
6.1.6.4 Important Features of Our Proposed Extension

The important features of our extension are listed below.

- It extends WSDL using XSD without compromising interoperability among services.
- Even, it does not affect loose coupling relationship of web services.
- It can handle structural dependencies introduced via semantic elements.
- Test-cases can be executed merely based on extended WSDL.
- It can be published with service registry UDDI.

6.1.6.5 WSDL Data Set Used in Experiments

As per our knowledge there does not exist any standardized benchmark WSDL data sets to validate our proposal. Due to non-availability of such benchmark models, we use some web service example deployed over the internet [242]. For the experiment, we selected some important WSDLs to test our proposed approach. The maximum WSDL file size was up to 901 Lines of Tag (LOT). However, currently, we carry out black-box testing based on static structural dependencies introduced through extended WSDL. We will further extend it to incorporate dynamic dependencies which arise during run-time invocation of web services.

6.1.6.6 Experimental Results

We have tested the working of our proposed approach using sample WSDLs stated in [242] with structural dependencies incorporated in WSDL. The system configuration used to carry out testing is windows 7 Professional service pack 1, intel(R) core(TM) i3-3240 CPU@3.40GHz running at 3.40 GHz, with 4.00 GB RAM. We studied the run-time requirements of our approach for these WSDL data sets and corresponding web services for several runs. All the measured times reported in this section are overall times, including building and deploying of extended WSDL. Table 6.1 summarizes the average run-time of web services. As we are not aware of the existence of any algorithm for WSDL-based testing, we have not presented any comparative results. We have presented only the results obtained from our experiments. Since, we black-box tested different WSDLs using their respective web service client and web services, we have calculated the average run-time requirements of corresponding web services. The performance results of our implementation are correct and satisfactory. From the experimental results, it can be observed that the average run-time
increases sublinearly as the lines of tag (LOT) increase in an extended WSDL as shown in Fig.6.18.

### Average run-time of web services using extended WSDL

![Average run-time of web services](image)

**FIGURE 6.18:** Average run-time of web services w.r.t line of tag

#### 6.1.7 Comparison With Related Work

We compare the performance of our approach with the existing approaches or techniques or frameworks for WSDL-based testing.

Tsai et al. [243] have proposed an XML-based object-oriented testing framework named *coyote* to test web services rapidly. The framework coyote consists of two parts: test master and test engine. The test master allows testers to specify test scenarios and test cases as well as various analyses such as dependency analysis, completeness and consistency and converts WSDL specifications into test scenarios. The test engine interacted with the web services under test and provides tracing information.

Bai et al. [250] have presented technique to generate web service test cases automatically based on the web services specification language web services description language (WSDL). The WSDL file is first parsed and transformed into the structured DOM tree. Then, test cases were generated from two perspectives: test data generation and test operation gen-
TABLE 6.1: Average run-time of web services

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Web Service Name</th>
<th>XMI (#LOT)</th>
<th>Average Run-Time (in Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>country</td>
<td>901</td>
<td>7.0</td>
</tr>
<tr>
<td>2</td>
<td>uszip</td>
<td>434</td>
<td>6.0</td>
</tr>
<tr>
<td>3</td>
<td>BibleWebService</td>
<td>412</td>
<td>5.0</td>
</tr>
<tr>
<td>4</td>
<td>airport</td>
<td>411</td>
<td>5.0</td>
</tr>
<tr>
<td>5</td>
<td>PeriodicTable</td>
<td>403</td>
<td>5.0</td>
</tr>
<tr>
<td>6</td>
<td>currency</td>
<td>315</td>
<td>4.0</td>
</tr>
<tr>
<td>7</td>
<td>GlobalWeather</td>
<td>238</td>
<td>3.0</td>
</tr>
<tr>
<td>8</td>
<td>MedicareSupplier</td>
<td>226</td>
<td>3.0</td>
</tr>
<tr>
<td>9</td>
<td>length</td>
<td>183</td>
<td>2.0</td>
</tr>
<tr>
<td>10</td>
<td>whois</td>
<td>162</td>
<td>2.0</td>
</tr>
<tr>
<td>11</td>
<td>creditcard</td>
<td>155</td>
<td>2.0</td>
</tr>
<tr>
<td>12</td>
<td>braile</td>
<td>154</td>
<td>2.0</td>
</tr>
<tr>
<td>13</td>
<td>isbn</td>
<td>153</td>
<td>2.0</td>
</tr>
<tr>
<td>14</td>
<td>RSSReader</td>
<td>152</td>
<td>2.0</td>
</tr>
<tr>
<td>15</td>
<td>Search [Section 5.1]</td>
<td>60</td>
<td>2.0</td>
</tr>
<tr>
<td>16</td>
<td>ProductRegistration [Section 5.1]</td>
<td>59</td>
<td>2.0</td>
</tr>
<tr>
<td>17</td>
<td>Login [Section 5.1]</td>
<td>58</td>
<td>2.0</td>
</tr>
<tr>
<td>18</td>
<td>CGCD (our example)</td>
<td>55</td>
<td>2.0</td>
</tr>
</tbody>
</table>
eration. Test data were generated by analyzing the message data types according to standard XML schema syntax. Operation flows are generated based on the operation dependency analysis. Three types of dependencies are defined: input dependency, output dependency, and input/output dependency. Finally, the generated test cases are documented in XML-based test files called service test specification.

Tsai et al. [237] have proposed extension to WSDL by introducing dependencies like input-output dependency, invocation sequence, hierarchical functional description and concurrent sequence specifications. Bartolini et al. [33, 34] automated WSDL-based testing, which combined the coverage of web service operations with data-driven test case generation. They defined a framework of a test environment that basically integrates two existing tools: soapUI [207], and TAXI [34]. The test suite generation is driven by basic coverage criteria and some heuristics.

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Related Work</th>
<th>Standard/ Technique/ Approach/ Framework Used</th>
<th>Dependency Used</th>
<th>Tool Used</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tsai et al. [243]</td>
<td>object-oriented testing framework</td>
<td>-</td>
<td>-</td>
<td>Test cases</td>
</tr>
<tr>
<td>2</td>
<td>Bai et al. [250]</td>
<td>WSDL</td>
<td>input, output and input/output</td>
<td>-</td>
<td>Service test case specification</td>
</tr>
<tr>
<td>3</td>
<td>Tsai et al. [237]</td>
<td>extended WSDL</td>
<td>input, output and input/output</td>
<td>-</td>
<td>Test cases</td>
</tr>
<tr>
<td>4</td>
<td>Bartolini et al. [33]</td>
<td>WSDL and XSD</td>
<td>-</td>
<td>soapUI and TAXI</td>
<td>Test cases</td>
</tr>
<tr>
<td>5</td>
<td>Bartolini et al. [34]</td>
<td>WS-TAXI framework</td>
<td>-</td>
<td>WS-TAXI</td>
<td>Test cases</td>
</tr>
<tr>
<td>6</td>
<td>Our approach</td>
<td>extended WSDL</td>
<td>control and data</td>
<td>NetBeans IDE</td>
<td>SOAP response</td>
</tr>
</tbody>
</table>

TABLE 6.2: Comparison with related work
A comparison between our work and some related work is presented in Table 6.2. Our approach for black-box testing of web services based on extended WSDL incorporates several new things as compared to other work reported in the literature. One new thing is that our testing approach is based on both data and control dependencies which is being reported in extended WSDL. Testing based on both these dependencies efficiently correlate different elements of WSDL during run-time of services.

### 6.2 Model-based Testing of Service-Oriented Software

Nowadays, many organizations are looking forward to increase the adaptability of their systems to react quickly to changes occurring in their environments. This requirement of system adaptability is best addressed by service-oriented architecture (SOA), as it facilitates the availability of environmental services provided by the third parties. This feature of SOA rapidly increases the adoption of SOA by different organizations. Hence, service-oriented architecture (SOA) is being used in design, development, deployment and management of different business services. These business services in SOA can also be used by other composite applications through publishing and binding interfaces. SOA integrates heterogeneous services and enables them for effectively exchanging information among member enterprises to process remote orders, inquiries, payroll, billing, HR and information delivery etc.

Due to the dynamic nature of SOA, its testing frequency is high. This results in an expensive process of test path construction and generation of test cases, each time an SOA-based software is tested. While testing SOA-based software, we have to generate test paths and test data values for each of the services involved in the business workflow. The assignment of test data values uses six different assertion types such as existing data value from the same interface, a constant value, a set of alternate values, a value range, a concatenated value and a computed value [210]. To enhance the efficiency of testing, wrong/faulty data plays a greater role. The use of faulty data in testing SOA-based software increases the fault tolerance of SOA-based software. In SOA, faulty data can be categorized into two sets with two different perspectives such as (i) to test the services in isolation and (ii) to test the service as a component in the system environment [264].

In this scenario, generation, and identification of test paths and test cases have become challenging factors as the raw code for web services and the associated test scripts are proprietary to the service providers and invisible to the tester. This situation can be avoided, if we generate test paths and test cases from the models describing SOA-based software.
6.2.1 Approach 1: Testing of Service-Oriented Software Using BPMN Diagram

In this section, we propose a model-based approach for testing SOA-based software using BPMN diagram. The business process modeling notation (BPMN) is a graphical notation that depicts the steps in a business process. It also depicts the end to end flow of a business process. The notation has been specifically designed to coordinate the sequence of business processes or services and the messages that flow between different processes or services present in a related set of activities [178]. The main advantages of using a BPMN model for testing of service-oriented software are as follows:

- It describes the services efficiently in order to analyze, and understand the SOA-based software.
- It also identifies which test paths need to be run to execute a particular business flow.

This BPMN model is usually designed and created in the early stage of software development life cycle. However, test case data generation can be made parallel to software development in order to reduce time and effort. Thus, testers will have time to pay attention to test the software before delivery.

A BPMN diagram contains four basic element categories (i) flow objects: these are the events, activities and gateways (ii) connecting objects: these are the sequence flow, message flow and association (iii) swim lanes: these are the pools and lanes (iv) artifacts: these are the data objects, group and annotations. These four categories enable the creation of simple business process diagrams (BPDs). Even BPDs also permit making new types of flow objects or artifacts, to make the diagram more understandable. This specification also includes semantic information related to SOA-based software. In this Section, we propose a method to test SOA-based software using BPMN diagram.

BPMN diagram is widely used in business process modeling (BPM). BPM is the activity of representing the enterprise processes so that, the current process may be analyzed and improved. BPMN model consists of simple diagrams constructed from a limited set of graphical elements such as flow objects, connecting objects, swimlanes, and artifacts. While BPMN shows the flow of data (messages), and the association of data artifacts to activities, it is not a data flow diagram. The BPMN diagram can also be used to test service-oriented software.
Our proposal for testing can be visualized as shown in Fig. 6.19. Below, we present our proposed approach as an algorithmic pseudo-code.

![Proposed testing approach diagram]

**Algorithm 2** Testing of Service-Oriented Software Using BPMN Diagram (TSOSBD)

1. *Model the Process*: design the BPMN diagram of the service-oriented software.
2. *Generate Control Flow Graph (CFG)*: assign version number to BPMN and unique node number to each service or task of BPMN. Then, generate the control flow graph of the BPMN model.
3. *Generate Test Paths*: apply depth first search (DFS) method for generating the test paths.
4. *Execute Test Cases*: apply test cases on generated test paths.

Let us follow the steps (represented by circles) one by one. The first step portrayed is *model the process*. This is an external task, in which a general task process, like travel reservation, shopping item etc. is being modeled using business process modeling notation (BPMN). Then, we export the BPMN file in \texttt{.bpmn} format. The XML DOM parser reads it and transforms its relevant data into a directed graph, this stage is called, as *Generate Control Flow Graph*. Then, the test paths are generated from CFG using depth first dearch (DFS) method. Finally, the test cases are executed on the generated test paths.

### 6.2.1.1 Model the Process

The service call specific control flow, leads us to choose a graph-based modeling approach for the business process. Each service or service call request in our model will be represented as an individual node. In the case of service-oriented software, where a service may call other
services can be represented as another node and an edge represents the control flow from one
service node to another service node. This inherent nature of service-oriented software leads
us to choose a graph-based modeling approach to represent the business processes. This also
has to do with the increased usage of business process modeling languages such as BPMN
[178] and WS-BPEL [179].

Most companies have already modeled their business processes with any one of these
two languages, which makes easy the model generation, often reduces execution costs and
effectively communicates the business processes in a standard manner. A BPMN defines
a business process diagram (BPD), based on a flowcharting technique tailored for creating
graphical models of business processes. The web service business process execution lan-
guage (WS-BPEL) [179], is an OASIS’s standard executable language for specifying actions
within business processes with web services. A business process model does not necessarily
have to be implemented as an automated business process in a process execution language.
Even by design, there are some limitations on the process topologies that can be described
in WS-BPEL, so it is possible to represent processes in BPMN that cannot be mapped to
WS-BPEL. Moreover, there are concepts, such as ad-hoc subprocesses, that BPMN can rep-
resent that may not be implemented with any technology [178]. Since there is a serious
advantage of BPMN and as it supports convenient graph-based modeling, we decided to use
BPMN with the help of Bizagi Modeler, a freeware tool [31]. The bizagi modeler is a model-
ing tool much like Microsoft Visio Professional [155] or Sparxsystems Enterprise Architect
12.1 [211], but more targeted towards the creation of BPMN models. It supports BPMN
model exchange through export or import with .bpmn format.

6.2.1.2 A BPMN Example: Online Shopping System (OSS)

The online shopping system (OSS) described in Section 5.1 can be represented as a set of
business services in a BPMN diagram as shown in Fig. 6.20. We have used Bizagi Modeler
tool [31] to design OSS. It shows various service tasks such as product registration,
courier company registration, login, sign up, third-party login, search product, add_to_cart,
make_payment and make_courier. These service tasks are being provided by a pool or more
precisely entities such as customer, payment gateway provider, online retailer, third-party lo-
gin provider, product seller, shipping and courier company. Other tasks are product registra-
tion request, courier company registration request, login request, sign up request, third-party
login request, search product request, make_payment request and make_courier request. The
information seen on the edges might represent one of the following two different things de-
pending on it’s source. If it is a service or task, it represents it’s output data, on the other
hand if it is a gateway, it contains the condition that makes the control flow in that direction.
FIGURE 6.20: A BPMN diagram for "online shopping system (OSS)"
Once the business process modeling is completed, it is exported in .bpmn format as shown in Fig. 6.21.

![Diagram of business process](image)

**FIGURE 6.21: Exporting BPMN with bizagi modeler**

### 6.2.1.3 Generate Control Flow Graph (CFG)

First, we briefly describe our CFG generation algorithm. Then, we present the pseudo-code of the algorithm. Subsequently, we discuss the complexity of our algorithm.

### 6.2.1.4 Overview of the CFG Generation Algorithm

Before execution of a business process, the CFG of a business process is constructed statically. It describes the sequence in which different *tasks* or *service tasks* of a business process get executed. Stating, in other words, a control flow graph (CFG) describes how the control flows through the business processes. Before presenting our proposed algorithm, few more definitions that would be used in our algorithm are introduced in this section.

**Definition 6.1 Unique Node Number Assignment:**

A unique node number is an incremental assignment of numeric numbers to each task of BPMN diagram.

**Definition 6.2 BPMN Version (V):**

Version (V) of a BPMN diagram comes into picture if an evolution happens in the BPMN diagram causing a change in the business flow of the system. This evolution may take ef-
fect due to the changes in the business service specification or the changes to the customer requirements. Once the changes have been incorporated, then the corresponding task of the BPMN diagram are updated and the BPMN diagram is redrawn resulting in a new version (say V1 to V2). For simplification of the proposed approach, we assume that the BPMN has initial version V1.

In the algorithm, we first create two special nodes \textit{start} and \textit{stop} corresponding to nodes \textit{startEvent} and \textit{endEvent} of BPMN model to define the start and end of business flow. Then after, we extract the root node \textit{process} and it’s subchild nodes \textit{task} or \textit{serviceTask} and create nodes in CFG and assign them unique node numbers and attributes. We add a control flow edge if \textit{textnode} value of \textit{outgoing} element of subchild \textit{n_i} equals \textit{textnode} value of \textit{incoming} element of subchild \textit{n_j}. We now present our CFG generation algorithm for our business process in the form of pseudo-code as below:

\textbf{Algorithm} : Control Flow Graph (CFG) Generation Algorithm.

\textbf{Input} : A BPMN model // in .bpmn format

\textbf{Output} : Control Flow Graph

1. \textbf{Node Construction}

   (a) Create two special nodes \textit{start} and \textit{stop} corresponding to nodes \textit{startEvent} and \textit{endEvent}

   (b) For each root node \textit{process} do the followings

   For each subchild node \textit{task} or \textit{serviceTask} node do the following

   i. Create node \textit{n_i}

   ii. Assign the node \textit{n_i} with unique node number.

2. \textbf{Add control flow edges}

   (a) For each root node \textit{process}, do the following

   For each subchild node \textit{task} or \textit{serviceTask} node do the following

   i. Add control flow edge from node \textit{(n_i,n_j)}, if \textit{textnode} value of \textit{outgoing} element of subchild \textit{n_i} equals \textit{textnode} value of \textit{incoming} element of subchild \textit{n_j}.
Fig. 6.22 shows the control flow graph of Fig. 6.20. The different numbered tasks serve as the nodes of the control flow graph. An edge from one node to another exists if the execution of the task representing the first node can result in the transfer of control to another node.
6.2.1.5  Complexity Analysis of CFG Generation Algorithm

Time complexity
To compute the time complexity of our CFG generation algorithm, we consider each step of the algorithm. Step 1(a) requires $O(1)$ constant time. Step 1(b)i requires $O(1)$ constant time, Step 1(b)ii requires $O(n^2)$ time and finally the Step 2(a)i requires $O(n^2)$ time. Hence the time complexity of our algorithm is $O(n^2)$, where $n$ is the input size.

Space complexity
The space complexity of our CFG generation algorithm would be $O(n)$, since XML DOM parser maintains a hash table for CFG, and the space complexity of every reasonable hash table is $O(n)$, where $n$ is the input size.

6.2.1.6  Implementation of CFG Generation Algorithm

As we said in the previous section, Bizagi Modeler exports a file with all the information we need to create a control flow graph. The XML DOM parser extracts and stores the BPMN elements in a data structure in such a way that it facilitates control flow graph generation, graph searches, and further test case generation. A BPMN file fragment is shown in Fig. 6.23. In order to search a service used in the BPMN diagram, the XML DOM parser traverses the DOM tree to find serviceTask element node. While traversing the services or tasks it also stores the attribute values of id, name and other elements such as incoming, outgoing, which are the element nodes needed for generating CFG. Fig. 6.24 shows the internal data structure maintained by XML DOM parser. The unique node number is assigned by the parser. The parser compares each outgoing element with incoming element to determine the possible control flows among the services or tasks. To easily describe the control flow from outgoing to incoming elements, we highlight the common ids with same color. The parser uses these elements and builds the CFG.

<serviceTask id="Id_25d2edf-2809-4678-965f-882a0b84ebc9" name="Product Registration Service">
  <extensionElements />
  <bizagi:BizagiExtensions xmlns:bizagi="http://www.bizagi.com/bpmn20">
    <bizagi:BizagiProperties>
      <Bizagi:BizagiProperty name="BgColor" value="#ECEFFF" />
      <Bizagi:BizagiProperty name="BorderColor" value="#036894" />
      <Bizagi:BizagiProperty name="runtimeProperties" value="{quot;cost;&quot;:0,&quot;prior" 
          </bizagi:BizagiProperties>
  </bizagi:BizagiExtensions>
</extensionElements>
  <incoming Id_8a26e5d1-5a85-4flb-9230-5338c60a6d10="incoming">
    <outgoing Id_007e221-1be7-4703-b604-302ee8210737=" outgoing" 
</serviceTask>

FIGURE 6.23: A BPMN file fragment
6.2.1.7 Generate Test Paths

Once we have generated the CFG, it is the time to generate test paths. Test paths can be generated by applying depth first search (DFS) method or breadth first search (BFS) method. In DFS, the path is generated by starting with the root node and exploring as far as possible along each branch before backtracking. In BFS, the path is generated by exploring all the nodes at one level completely and then moving into the next level and so on. The limitation of BFS is that it can lead to generate an exponential number of test paths which will increase the time complexity. The paths to be tested are coming out in exponential, out of which very few are useful, even there is a need of testing only basic paths. With this problem, and considering the efficiency of DFS method to generate test path, we use DFS method to generate test paths from BPMN diagram. By using this approach, the useless test paths are eliminated which in turn reduces the time complexity. The generated test paths for the CFG given in Fig. 6.22 using DFS are given below:

1. start → 1 → 2 → stop.

2. start → 3 → 4 → 13 → 15 → 16 → 17 → stop.

3. start → 7 → 8 → 13 → 15 → 16 → 17 → stop.

4. start → 9 → 10 → 13 → 15 → 16 → 17 → stop.

5. start → 11 → 12 → 13 → 15 → 16 → 17 → stop.
6. start $\rightarrow$ 5 $\rightarrow$ 6 $\rightarrow$ stop.

7. start $\rightarrow$ 3 $\rightarrow$ 4 $\rightarrow$ 13 $\rightarrow$ 15 $\rightarrow$ 14 $\rightarrow$ 15 $\rightarrow$ 16 $\rightarrow$ 17 $\rightarrow$ stop.

8. start $\rightarrow$ 7 $\rightarrow$ 8 $\rightarrow$ 13 $\rightarrow$ 15 $\rightarrow$ 14 $\rightarrow$ 15 $\rightarrow$ 16 $\rightarrow$ 17 $\rightarrow$ stop.

9. start $\rightarrow$ 9 $\rightarrow$ 10 $\rightarrow$ 13 $\rightarrow$ 15 $\rightarrow$ 14 $\rightarrow$ 15 $\rightarrow$ 16 $\rightarrow$ 17 $\rightarrow$ stop.

10. start $\rightarrow$ 11 $\rightarrow$ 12 $\rightarrow$ 13 $\rightarrow$ 15 $\rightarrow$ 14 $\rightarrow$ 15 $\rightarrow$ 16 $\rightarrow$ 17 $\rightarrow$ stop.

**6.2.1.8 Execute Test Cases**

The main goal of this step is to generate test case data, and execute the test cases for each business process based on the generated test paths.

We choose a path coverage-based test case generation strategy where test cases are designed such that all *linearly independent paths* in the business process are executed at least once. A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent path [194].

Our goal is to provide semi-automatically input values which will exercise each linearly independent path previously created. To test a specific path, a number of service tasks or tasks need to be executed. In regular cases, most service tasks or tasks require some type of input data. The only way to do it is by whitening up a bit for testing. In other terms, the service providers will have to give a description of what are the possible inputs and outputs of a service task or tasks. Let the service provider declares inputs, data types and restrictions for each of the service or task as shown in Table 6.3. When we have sufficient information of tasks or service tasks, we can discover what inputs will fit, i.e., what input data will produce results matching those restrictions and corresponding test paths.

Once we have such information as provided in Table 6.3 we can supply test cases in such a way that each identified test path is covered. For example, we can exercise test path 1 by supplying test case set \{product name=mi3, quantity=1, price=12000, seller name=xiaomi, contact address=china\} as input, and observe the test case status as valid. This test path covers the unique node numbers 1 and 2 of CFG. Similarly, we can test all the test paths by executing appropriate test cases and check the output. These test cases along with their status are shown in Table 6.4.
TABLE 6.3: Inputs, data types and restrictions of service or task

<table>
<thead>
<tr>
<th>Unique Node Number</th>
<th>Element Name</th>
<th>Data Type</th>
<th>XSD Constraints</th>
<th>XSD Constraints Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &amp; 2</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>quantity</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>price</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>seller name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>contact address</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>3 &amp; 4</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>3 &amp; 4</td>
<td>pincode</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>3 &amp; 4</td>
<td>quantity</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>shipping company name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>address</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>charges</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>7 &amp; 8</td>
<td>username</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>7 &amp; 8</td>
<td>password</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>9 &amp; 10</td>
<td>mobile no</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
</tr>
<tr>
<td>11 &amp; 12</td>
<td>username</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>11 &amp; 12</td>
<td>tpassword</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>quantity</td>
<td>xs:int</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>customer name</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>address</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>contact no</td>
<td>xs:int</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>card type</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>card no</td>
<td>xs:int</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>expiry date</td>
<td>xs:date</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>PIN</td>
<td>xs:int</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>OTP</td>
<td>xs:int</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>customer name</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>address</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccures</td>
<td>1</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>contact no</td>
<td>xs:int</td>
<td>minOccures</td>
<td>1</td>
</tr>
</tbody>
</table>
### TABLE 6.4: Test cases with output for the example “online shopping system (OSS)”

<table>
<thead>
<tr>
<th>Unique Node Number</th>
<th>Element Name</th>
<th>Data Type</th>
<th>XSD Constraints</th>
<th>XSD Constraints Value</th>
<th>Input</th>
<th>Test Case Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &amp; 2</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>mi3</td>
<td>Valid</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>quantity</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>mi3</td>
<td>Invalid</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>price</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>12000</td>
<td>Valid</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>seller name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>1</td>
<td>Invalid</td>
</tr>
<tr>
<td>1 &amp; 2</td>
<td>contact address</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>china</td>
<td>Valid</td>
</tr>
<tr>
<td>3 &amp; 4</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>1</td>
<td>Invalid</td>
</tr>
<tr>
<td>3 &amp; 4</td>
<td>pincode</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>384001</td>
<td>Valid</td>
</tr>
<tr>
<td>3 &amp; 4</td>
<td>quantity</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>xiaomi</td>
<td>Invalid</td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>shipping company name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>DHL</td>
<td>Valid</td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>address</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>1</td>
<td>Invalid</td>
</tr>
<tr>
<td>5 &amp; 6</td>
<td>charges</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>5000</td>
<td>Valid</td>
</tr>
<tr>
<td>7 &amp; 8</td>
<td>username</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>ABC</td>
<td>Valid</td>
</tr>
<tr>
<td>7 &amp; 8</td>
<td>password</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>123</td>
<td>Valid</td>
</tr>
<tr>
<td>9 &amp; 10</td>
<td>mobile no</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>9725577940</td>
<td>Valid</td>
</tr>
<tr>
<td>11 &amp; 12</td>
<td>tusername</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>XYZ</td>
<td>Valid</td>
</tr>
<tr>
<td>11 &amp; 12</td>
<td>tpassword</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>test123</td>
<td>Valid</td>
</tr>
<tr>
<td>13</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>mi3</td>
<td>Valid</td>
</tr>
<tr>
<td>13</td>
<td>quantity</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>1</td>
<td>Valid</td>
</tr>
<tr>
<td>13</td>
<td>customer name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>KKR</td>
<td>Valid</td>
</tr>
<tr>
<td>13</td>
<td>address</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>india</td>
<td>Valid</td>
</tr>
<tr>
<td>13</td>
<td>contact no</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>9726677988</td>
<td>Valid</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>card type</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>CREDIT</td>
<td>Valid</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>card no</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>112233</td>
<td>Valid</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>expiry date</td>
<td>xs:date</td>
<td>minOccurs</td>
<td>1</td>
<td>2020-05-05</td>
<td>Valid</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>PIN</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>112233</td>
<td>Valid</td>
</tr>
<tr>
<td>14 &amp; 15</td>
<td>OTP</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>1334</td>
<td>Valid</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>customer name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>KKR</td>
<td>Valid</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>address</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>india</td>
<td>Valid</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>product name</td>
<td>xs:string</td>
<td>minOccurs</td>
<td>1</td>
<td>mi3</td>
<td>Valid</td>
</tr>
<tr>
<td>16 &amp; 17</td>
<td>contact no</td>
<td>xs:int</td>
<td>minOccurs</td>
<td>1</td>
<td>9726677988</td>
<td>Valid</td>
</tr>
</tbody>
</table>
6.2.2 Approach 2: Testing of Service-Oriented Software Using SoaML Service Interface Diagram

This section also proposes an approach to test service-oriented software. Our proposed testing approach uses SoaML service interface diagram. To illustrate our proposed approach, we consider the service-oriented software case study “online shopping system (OSS)” described in Section 5.1. The detailed steps of our proposed testing approach is given below:

**Algorithm 3 Testing of Service-Oriented Software Using SoaML Service Interface Diagram (TSOSSID)**

1. Design *SoaML service interface diagram* for the given service-oriented software.
2. Convert the *SoaML service interface diagram* into its equivalent XML schema.
3. Generate XML instances of each service interface present in the XML schema.
4. Apply various XML schema definition (XSD) constraints to the generated instances to test the service-oriented software.

6.2.2.1 Design of SoaML service interface diagram and generation of XML Schema

The first step of our approach is to design the service interface diagram. For this purpose, we use the tool, Visual Paradigm [230]. The service interface diagram for “online shopping system (OSS)” is shown in Fig. 6.25. Using the same tool, Visual Paradigm, we also perform the second step of our approach which generates the equivalent XML schema for the services described in the service interface diagram as shown in Fig. 6.26.

![SoaML service interface diagram for “online shopping system (OSS)”](image)
The generated XML schema for the SoaML service interface diagram is shown in Fig. 6.27. 

FIGURE 6.26: XML schema or instant generation for “online shopping system (OSS)”

FIGURE 6.27: Generated XML schema for “online shopping system (OSS)”
6.2.2.2 Generating Instances in XML schema

After generating the equivalent XML schema from the service interface diagram, we generate an instance for each service interface present in the XML schema. The XML schema has mainly two service interfaces consumer and provider which are represented in the service interface diagram. To generate an instance of each of this interface we insert an XML element instance as of type online_shopping_service, which is a global type as shown in Fig. 6.28 in red colour.

FIGURE 6.28: Instance generation in XML schema

This schema instance or subschema gives a simplified view of the attributes (data) and
operations of the service interfaces. It also enables the tester to test relations among the service interfaces by including the XML elements. For example, the XML schema of consumer and provider under the parent element online_shopping_service gives information that there is a relation between the consumer and provider interfaces.

The XML instance will now become a part of the WSDL document. When this sub-schema is passed through Altova XMLSpy [13], it generates a representative test case for XML instance as shown in Fig. 6.29.

FIGURE 6.29: A sample test case for XML schema instance

Fig. 6.29 shows the sample test cases in the required form. In Fig. 6.28, the XML element <xs:element name="seller_name" type="xs:string" minOccurs="0" maxOccurs="1"> shows the type of data that is generated or required to be generated. When the code shown in Fig. 6.28 is passed through Altova XMLSpy, the final XML file is generated, which has automatically generated test cases as shown in Fig. 6.29. After generating the test cases for XML instances now we are able to build a test case data set. These test case data sets can be used in the generated XML instance of the service interface diagram to further generate test cases by applying XSD constraints.
6.2.2.3 Executing Test Cases by Applying XML Schema Constraints

In SOA-based software, the XSD can be used to express a set of rules to which a WSDL or XML document must confirm in order to be considered “valid” according to that schema. However, unlike most other schema languages, XSD was designed with the intent that determination of a document’s validity would produce a collection of information adhering to specific data types [255].

The XML schema datatype may be simple datatype, complex datatype or built-in datatype. Simple datatype is defined with tag `<xs : simpleType.. >`. Complex datatype is defined with tag `<xs : complexType..>`.

Complex datatype includes sequence, choice or all sub-elements. The sequence sub-element displays in same order i.e. child elements can only appear in the sequence of order that is already mentioned, here no randomization is allowed. In XSD all we can have all the child elements in the order we may like. These schema data types once defined along with XSD constraints, their values appear in the XML instances. We can find these links by considering Fig. 6.27 and Fig. 6.28 respectively. The choice sub-element indicates that there is only one sub-element to appear. If the sub-element appears under all category, then it may be presented in any order. However, the built-in datatype is defined in the XML schema. We can use various XSD constraints in the XML instances to customize our test cases.

We can use various XSD constraints in the XML instances to customize our test cases. `maxLength`, `minLength`, `maxExclusive`, `minInclusive`, `whitespace`, and `enumeration` etc. are some examples of XSD constraints. For example, `maxLength` is used to specify the maximum number of characters for `char`. Similarly, `minLength` specifies the minimum number of characters. The `maxInclusive` specifies the upper bound whereas `minInclusive` specifies the lower bound. `whitespace` can be used to handle spaces, tab and line feed. `Enumeration` constraint defines the acceptable value list. We can customize our test cases by applying various XSD constraints in XML instances. XML schema defines various built-in data types, such as `xs:int`, `xs:float`, `xs:double`, `xs:decimal`, `xs:date`, `xs:time`, `xs:long`, `xs:short`, etc. We can change the identity of the data type to generate various test cases. For example, we can change the `xs:string` data type into `xs:int` to generate the `minValue` and `maxValue` of the XML schema. Here, we generate the test cases for the service interfaces `consumer` and `provider`. The XML schema and the instance for OSS are shown in Fig. 6.27 and Fig. 6.28. By putting different test case data, we generate different test cases for our example online shopping system (OSS). Some of the valid and invalid test cases, which we have obtained for our case study OSS, are shown in Table 6.5. To demonstrate the working of our approach, test inputs like xiaomi, mi, 12, 12345 etc. are quantitative evaluation parameters which are executed and test results are recorded.
TABLE 6.5: Test case execution results for “online shopping system (OSS)” using SoaML service interface diagram

<table>
<thead>
<tr>
<th>Test Case ID</th>
<th>Element Name</th>
<th>Data Type</th>
<th>Input</th>
<th>XSD Constraints</th>
<th>Expected Test Case Result</th>
<th>Actual Test Case Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>seller_name</td>
<td>xs:string</td>
<td>xiaomi</td>
<td>minOccurs 0</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>2</td>
<td>seller_name</td>
<td>xs:string</td>
<td>xiaomi</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>3</td>
<td>seller_name</td>
<td>xs:string</td>
<td>12</td>
<td>minOccurs 1</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>4</td>
<td>product_name</td>
<td>xs:string</td>
<td>mi</td>
<td>minOccurs 0</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>5</td>
<td>product_name</td>
<td>xs:string</td>
<td>mi</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>6</td>
<td>product_name</td>
<td>xs:string</td>
<td>12345</td>
<td>minOccurs 1</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>7</td>
<td>quantity</td>
<td>xs:int</td>
<td>12345</td>
<td>minOccurs 0</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>8</td>
<td>quantity</td>
<td>xs:int</td>
<td>12345</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>9</td>
<td>quantity</td>
<td>xs:int</td>
<td>ABC</td>
<td>minOccurs 1</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>10</td>
<td>contact_address</td>
<td>xs:string</td>
<td>China</td>
<td>minOccurs 0</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>11</td>
<td>contact_address</td>
<td>xs:string</td>
<td>China</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>12</td>
<td>contact_address</td>
<td>xs:string</td>
<td>123</td>
<td>minOccurs 1</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>13</td>
<td>price</td>
<td>xs:int</td>
<td>12000</td>
<td>minOccurs 0</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>14</td>
<td>price</td>
<td>xs:int</td>
<td>12000</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>15</td>
<td>price</td>
<td>xs:int</td>
<td>XYZ</td>
<td>minOccurs 1</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>16</td>
<td>shipping_company_name</td>
<td>xs:string</td>
<td>DHL</td>
<td>minOccurs 0</td>
<td>Invalid</td>
<td>Invalid</td>
</tr>
<tr>
<td>17</td>
<td>shipping_company_name</td>
<td>xs:string</td>
<td>DHL</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>19</td>
<td>address</td>
<td>xs:string</td>
<td>DEF</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>20</td>
<td>username</td>
<td>xs:string</td>
<td>test</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>21</td>
<td>tpassword</td>
<td>xs:string</td>
<td>test123</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>22</td>
<td>username</td>
<td>xs:string</td>
<td>test1</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>23</td>
<td>password</td>
<td>xs:string</td>
<td>test123456</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>24</td>
<td>pincode</td>
<td>xs:int</td>
<td>4</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>25</td>
<td>card_type</td>
<td>xs:string</td>
<td>MAESTRO</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>26</td>
<td>card_no</td>
<td>xs:int</td>
<td>123456789</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>27</td>
<td>expiry_date</td>
<td>xs:date</td>
<td>2016-05-24</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>28</td>
<td>PIN</td>
<td>xs:int</td>
<td>4554</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>29</td>
<td>OTP</td>
<td>xs:int</td>
<td>4664</td>
<td>minOccurs 1</td>
<td>Valid</td>
<td>Valid</td>
</tr>
</tbody>
</table>
6.2.3 Comparison With Related Work

In this section, we compare our work with the existing work based on models.

Nonchot et al. [46] have focused on test case generation using business process modeling notation (BPMN), business process execution language (BPEL), and XSD schema (XSD), while our Approach 1 uses BPMN, and XSD constraints and Approach 2 uses SoaML service interface diagram which is a domain specific model for SOA interfaces, and uses XSD constraints to generate test cases. They [46] merged BPMN diagram and BPEL diagram and showed the manually generated control flow graph (CFG) while our Approach 1 produces automatically the intermediate representation, a control flow graph (CFG) using our CFG generation algorithm and our Approach 2 automatically generates XSD instances using Visual Paradigm [230]. Further, we demonstrated working of our Approach 1 using Altova XMLSpy [13], which automatically generated sample test cases from XSD schema, while they [46] did not.

Kumar et al. [21] have presented an automatic test data generation algorithm for web services through XML schema (XSD) and using TAXI [34]. They converted XML schema (XSD) into tree table of XML schema (XSD) to make it suitable for TAXI [34] to generate test data, while in our Approach 1 we used BPMN, XSD constraints, CFG, and a tool Altova XMLSpy [13] for test case execution. Further, the tool Altova XMLSpy [13] does not have such stringent requirement of input such as TAXI [34] has, it accepts XSD schema (XSD) and generates sample test cases. Even our Approach 2 does not require such transformation of inputs.

Hou et al. [213] have converted business process execution language (BPEL) processes to an intermediate representation, message sequence graph (MSG) to generate test cases, while our Approach 1 generates efficient intermediate representation, a control flow graph (CFG) of a BPMN model through our own algorithm. Our Approach 2 did not use such intermediate forms and used SoaML service interface model. Further, they [213] carried out a BPEL testing, more concretely a white box testing, while our both approaches are model based.

Mao [44] have computed static slices of a business process execution language (BPEL) programs for understanding and debugging purpose. While our Approach 1 tests the business process using BPMN model and Approach 2 did testing with the help of SoaML service interface model and XSD constraints. They [44] proposed a new intermediate presentation called BPDG, by identifying data, control, and synchronized dependencies in a business process execution language (BPEL) programs through static analysis, we generated a control flow graph (CFG) in our Approach 1 through BPMN model analysis and an instance of
XSD through SoaML service interface model. They [44] have presented a static (backward / forward) slicing algorithm which computes forward and backward slices for debugging BPEL program, while we apply XSD constraint in both approaches to generate test cases. Similarly, Endo et al. [17] have proposed a new graph called as parallel control flow graph (PCFG) which represents multiple CFGs of service composition and interactions among these CFGs.

Yee [45] have presented a test case generation tool for business process execution language (BPEL) using XPath expression and Type definitions of WSDL document, while in our Approach 1, we have used BPMN model, CFG and XSD to execute test cases and in Approach 2, we have used SoaML service interface model, and XSD. Our both approaches test web services based on the model, while they developed a testing tool for service compositions using business process execution language (BPEL) process.

Yuan et al. [258], have manually generated test cases from business process execution language (BPEL) code by converting it into an intermediate form named as business flow graph (BFG), while we used BPMN model and SoML service interface diagram in our Approach 1 and Approach 2 respectively. Their approaches [258] used service composition i.e. BPEL, while we unit tested web services based on models only.

Linzhang et al. [248] have used UML activity diagram to derive test scenarios and testing paths while in our Approach 1, we used BPMN, control flow graph (CFG), path coverage and XSD to test all the scenarios and in Approach 2 we have used SoaML service interface diagram, instance generation, and XSD constraints to test all the testing paths.

Gupta [124] has automatically generated test scenarios using UML activity diagram and depth search (DFS) method, while we used BPMN and SoaML service interface diagram in our approaches. He [124] has generated intermediated forms such as activity flow graph (AFG) and activity dependency table (ADT) automatically for each activity diagram while in our Approach 1 we have developed an efficient intermediate representation which we named as control flow graph (CFG) for web services.

Bhattacharjee et al. [82] have generated test cases based on UML activity diagram and tabu search algorithm to prioritize the test paths, while we developed our own algorithm for control flow graph (CFG) generation in our Approach 1, which further facilitated the testing process. Kaur et al. [186] have used UML statechart diagrams, a testing tool TestOptimal, chinese postman and prefix-based algorithms to automatically and manually derive test paths using graph coverage techniques, while we have used BPMN model, XSD, Altova XMLSpy [13], and path coverage to test web services in our Approach 1 and SoaML service interface model and Visual Paradigm [230] to test web services in our Approach 2. A comparison of our work and some related work is presented in Table 6.6.
<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Related Work</th>
<th>Model/ Language Used</th>
<th>Intermediate Representation Used</th>
<th>Testing Based On</th>
<th>Tool Support</th>
<th>Degree of Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nonchot et al. [46]</td>
<td>BPEL, BPMN</td>
<td>-</td>
<td>XSD</td>
<td>-</td>
<td>Fully</td>
</tr>
<tr>
<td>2</td>
<td>Kumar et al. [21]</td>
<td>XSD</td>
<td>XML Tree Table</td>
<td>XSD</td>
<td>TAXI [34]</td>
<td>Fully</td>
</tr>
<tr>
<td>3</td>
<td>Hou et al. [213]</td>
<td>BPEL</td>
<td>Message Sequence Graph</td>
<td>-</td>
<td>-</td>
<td>Manual</td>
</tr>
<tr>
<td>4</td>
<td>Mao [44]</td>
<td>BPEL</td>
<td>Extended Control Flow Graph</td>
<td>-</td>
<td>-</td>
<td>Fully</td>
</tr>
<tr>
<td>5</td>
<td>Endo et al. [17]</td>
<td>BPEL</td>
<td>Parallel Control Flow Graph</td>
<td>-</td>
<td>-</td>
<td>Semi</td>
</tr>
<tr>
<td>6</td>
<td>Yee [45]</td>
<td>BPEL</td>
<td>-</td>
<td>-</td>
<td>WSDL</td>
<td>Manual</td>
</tr>
<tr>
<td>7</td>
<td>Yuan et al. [258]</td>
<td>BPEL</td>
<td>Business Flow Graph</td>
<td>-</td>
<td>-</td>
<td>Manual</td>
</tr>
<tr>
<td>8</td>
<td>Linzhang et al. [248]</td>
<td>UML Activity Diagram</td>
<td>-</td>
<td>-</td>
<td>UMLTGF</td>
<td>Manual</td>
</tr>
<tr>
<td>9</td>
<td>Gupta et al. [124]</td>
<td>UML Activity Diagram</td>
<td>Activity CFG, Activity Dependency Table</td>
<td>Path Coverage</td>
<td>-</td>
<td>Semi</td>
</tr>
<tr>
<td>10</td>
<td>Bhattacharjee et al. [82]</td>
<td>UML Activity Diagram</td>
<td>-</td>
<td>Tabu Search Algorithm</td>
<td>-</td>
<td>Manual</td>
</tr>
<tr>
<td>11</td>
<td>Kaur et al. [186]</td>
<td>UML Statechart Diagram</td>
<td>-</td>
<td>Node, Edge, Edge pair, Prime Path Coverage</td>
<td>TestOptimal</td>
<td>Fully</td>
</tr>
<tr>
<td>12</td>
<td>Our Approach 1</td>
<td>BPMN</td>
<td>CFG</td>
<td>XSD, Path Coverage</td>
<td>Altova XMLSpy [13]</td>
<td>Semi</td>
</tr>
<tr>
<td>13</td>
<td>Our Approach 2</td>
<td>SoaML Service Interface Diagram</td>
<td>XML Instances</td>
<td>XSD</td>
<td>Visual Paradigm [230]</td>
<td>Semi</td>
</tr>
</tbody>
</table>
6.2.4 Conclusion

In this chapter, we have successfully applied black-box testing strategy to test web services. The novelty of our approach is in imposing a hierarchical structure on WSDL, getting dependency information in the form of extensible elements. We have successfully tested our approach. Our testing approach of extension to WSDL can further be improved by getting run-time information to carry out white-box testing.

This chapter also illustrated the testing approaches for SOA-based software using BPMN and SoaML diagrams. First, we described our testing algorithm for service-oriented software using BPMN diagram. In that we presented a CFG generation algorithm and its implementation. Next, we tested the test paths of CFG by applying various test cases. Secondly, we presented another algorithm for testing service-oriented software using SoaML service interface diagram. In that context, we generated XML schema and its instance. Finally, we tested the XML schema instance.