CHAPTER - 3

Review of Related Work

This chapter discusses the work done by previous researchers on static slicing of object-oriented software, dynamic slicing of object-oriented software, and black-box testing of SOA-based software. As we could not find any reported work in the literature on static and dynamic slicing of SOA-based software, so in this chapter, first we briefly discuss static and dynamic slicing of object-oriented software. Then, we review the work reported on black-box testing of SOA-based software. Finally, we describe the available work on model-based testing of SOA-based software.

3.1 Static Slicing of Object-Oriented Software

To the best of our knowledge, no work has been done on static slicing of SOA-based software. So, in this section, we discuss the work reported on static slicing of object-oriented software.

A natural way of localizing an error is to consider only those statements of a program, which tend to the erroneous behaviour being observed. Often this results in finding the statements of a program relevant to the value of a chosen variable at a given location of that program. This approach is called program slicing. The given variable and location form the slicing criterion. The original concept of program slicing was proposed by Mark Weiser [140] as another approach for debugging sequential programs. He claims that program slicing corresponds to the mental abstraction performed by programmers while debugging programs. Agrawal et al. [86] have presented a uniform approach to compute dynamic slices of programs that may involve unconstrained pointers, composite variables, and procedures. Program slicing is an active area of research, and this is reflected in various surveys such as in [69].

In 1995, Sun Microsystems Inc. introduced a new programming language called “Java”. The Sun Microsystems Inc., now Oracle and Sun Microsystems Inc. defined Java as “simple, object-oriented, network-savvy, interpreted, robust, secure, architecturally neutral, portable, high-performance, multithreaded and dynamic language. Java
has become the standard language for sending platform independent program over the internet and executing them as part of web pages.

Bik et al. [10] worked on restricting compiler that converts implicit parallelism in Java into explicit parallelism. Zhao [116] gave solution for static slicing concurrent Java programs. Nanda et al. [139] gave issues and solution for slicing concurrent Java programs, their work was inspired from [116]. Nanda et al. [139] described various algorithms for slicing concurrent inter-procedural and intra-procedural Java programs and gave formal proof of all the algorithms.

Catano [170] described a framework to slice Java event space for Java memory model. He defined Java event space as the partial orders of the memory and thread actions generated by a multithreaded Java program. He suggested an approach to reduce the size of Java event space and described the problem of aliasing that arises when two or more variables of the event space point of the same memory address. He suggested an algorithm to deal with it.

Lallchandani et al. [99, 110] have presented static slicing of UML architectural model by merging class diagram with a sequence diagram. They defined various dependencies like member, method, data, call, return, attribute, control, inter-sequence call and inter-sequence return. They extended the work presented in [131]. Zhao [100] introduced static slicing technique, considering ACME ADL for dependence analysis of software architecture.

### 3.2 Dynamic Slicing of Object-Oriented Software

In this section, we briefly present the reported work on dynamic slicing, architectural and UML model based slicing of object-oriented software. Most of the work reported in the literature are focused on the development of techniques for slicing UML models like class diagram, sequence diagram, use case diagram, activity diagram etc.

Korel and Laski [29] introduced a new form of slicing. This new form of slicing is dependent on input data and is generated during execution-time analysis as opposed to Weiser’s static slicing [140] and is therefore called dynamic slicing. Similar to major objective of static slicing, dynamic slicing was specifically designed as an aid to debugging, and can be used to help in the search for offending statements which caused the program error [128].

Considerable research results on dynamic slicing of procedural programs are available [29, 147]. But dynamic slicing of object-oriented programs have scarcely been reported in the literature [69, 140]. Agrawal and Horgan [22] were the first to present
algorithms for finding dynamic slices of procedural programs using program dependence graphs. They have proposed a dynamic slicing method by marking nodes on a static program dependence graph. The computed slice is not always precise, because some dependencies might not hold in dynamic execution. They also proposed a precise method based on the dynamic dependence graph (DDG) [22]. Zhao [269] extended the DDG of Agrawal and Horgan [22], known as dynamic object-oriented dependence graph (DODG) to represent various dynamic dependencies between statement instances for a particular execution of an object-oriented program. The DODG is an arc-classified diagraph \((V, A)\), where \(V\) is the multi-set of flow-graph vertices, and \(A\) is the set of arcs representing dynamic control dependencies and data dependencies between vertices. His construction of the DODG is based on dynamic analysis of control flow and data flow of the program, and similar to those for constructing dynamic dependence graphs for procedural programs [23]. They have constructed the DODG by creating a new node for each occurrence of a statement in the execution history, and creating all the dependence edges associated with the occurrence at run-time. He has also considered the specific features of object-oriented programs such as method calls, inheritance and polymorphism. The DODG of the example program in Fig. 3.1 on input \(argv[1] = 3\), is shown in Fig. 3.2.

Zhao has adopted the following concepts for dynamic slicing of object-oriented programs:

- A **slicing criterion** for an object-oriented program \(P\) is of the form \((s, v, t, i)\), where \(s\) is a statement in the program, \(v\) is a variable used at \(s\), and \(t\) is an execution trace of the program with input \(i\).

- A **dynamic slice** of an object-oriented program \(P\) on a given slicing criterion \((s, v, t, i)\) consists of all statements in the program that actually affected the value of a variable \(v\) at statement \(s\), for the given input \(i\).

Based on the DODG, Zhao has used a two-phase algorithm to compute dynamic slices of object-oriented programs. Computation of dynamic slices using the DODG is carried out as a graph-reachability problem. The two phases of the algorithm are:

(a) Computing a dynamic slice over the DODG of the object-oriented program. (This can be done by using a usual depth-first or breadth-first graph traversal algorithm to traverse the DODG of the program by taking the vertex corresponding to the statement of interest as the start point of traversal.)

(b) Mapping the slice over the DODG to the source code to obtain a dynamic slice of the program. (This can be done by simply defining a mapping function.)
1: class Elevator{
   public:

2:     Elevator(int ) /* initialization for Elevator */
3:         { current_floor = 1;  
4:             current_direction = UP;
5:             top_floor = 1;  }  /* end of Elevator */
6:     virtual ~Elevator() {}
7:     void up()
8:         { current_direction = UP; }
9:     void down()
10:         { current_direction = DOWN; }
11:     int which_floor()
12:         { return current_floor; }
13:     Direction direction()
14:         { return current_direction; }
15:     virtual void go(int floor) /* declaration for method go */
16:         { if (current_direction = UP)
17:             { while (current_floor != floor)
18:                 add(current_floor, 1);  }
19:          else
20:             { while (current_floor != floor)
21:                 add(current_floor, -1);  }  /* end */
22:         }
23:     private:
24:         int &a, const int &b) /* This method computes value of current_floor */
25:         { a = a + b;  }  
26:     protected:
27:         int current_floor;
28:     Direction current_direction;
29:     int top_floor;
30:     public:
31:     class AlarmElevator: public Elevator { /* AlarmElevator is derived from Elevator */
32:         public:
33:             AlarmElevator(int top_floor);
34:             Elevator(int top_floor)
35:     }
36:         void set_alarm()
37:     }
38:         void reset_alarm()
39:     }
40:         void go(int floor)
41:             { if (!alarm_on)
42:                 Elevator::go(floor);  
43:     }
44:         protected:
45:         int alarm_on;
46:     }
47:         Elevator *c_ptr;
48:         if (argc) c_ptr = new Elevator(10);
49:     else
50:         c_ptr = new AlarmElevator(10);
51:         c_ptr->go(3); /* polymorphic method call */
52:         cout << "\n currently on floor:";
53:         << c_ptr->which_floor();
54:     }  /* end of main */
55: }
56:  /* main */
57: } /* end of program */

FIGURE 3.1: An example program
Mohapatra [55] have presented work concerning dynamic slicing of object-oriented programs. He developed intermediate representation *extended system dependence graph* (ESDG). Then, he presented dynamic slicing algorithm *edge marking dynamic slicing* (EMDS) using ESDG. The EMDS algorithm is based on marking and unmarking the edges of the ESDG as and when dependencies arise and cease during run-time. EMDS algorithm marks an edge of the ESDG when its associated dependence exists and unmarks an edge when the dependence ceases to exist. Then, he presented another dynamic slicing algorithm *node marking dynamic slicing* (NMDS) using ESDG as the intermediate representation. The NMDS algorithm is based on marking and unmarking the nodes of the ESDG appropriately during run-time. He had shown that each of his proposed algorithm is more efficient than the related algorithms [269]. Next, he extended intermediate representation (ESDG) to *concurrent object-oriented programs*. He named this intermediate representation as *concurrent system dependence graph* (CSDG). The CSDG represents the concurrency aspects such as *synchronization dependencies* and *communication dependencies* of object-oriented programs. Then, he extended edge marking dynamic slicing (EMDS) algorithm to handle concurrency issues in object-oriented programs. He named algorithm as *marking based dynamic slicing* (MBDS). The MBDS algorithm handles *synchronization dependencies* and *communication dependencies* while computing dynamic slices.

Zhao [101] was the first person to introduce the concept of architectural slicing based on architectural description language (ADL) ACME. He defined component-connector
dependency, connector component dependency, and additional dependency. He had proposed an algorithm to compute architectural slice based on software architectural dependency graph (SADG). He extended his previous work [100] by introducing architectural information flow graph with information flow arcs like component-connector, connector-component, and internal flow arcs based on WRIGHT ADL. Kim et al. [217] have introduced dynamic software architecture slicing (DSAS) as a set of architect components and connectors that are relevant to the particular variable and events of interest at some point during the execution of software architecture. Korel et al. [25] have presented deterministic and nondeterministic slicing based on extended finite state machines (EFSMs). They have developed a tool to demonstrate their slicing technique. Kagdi [92] have introduced the concept of model slice, which extracts slices from a class diagram.

Samuel et al. [190] have presented a technique to test object-oriented software using dynamic slicing of UML sequence diagram. They used message guards on sequence diagram to generate dynamic slice with respect to each conditional predicates. Slices were generated from the dependency graph for all the variables at each message point in the sequence diagram.

Lallchandani et al. [99, 131] have used generic class diagram and generic sequence diagram and integrated them to generate a model dependency graph (MDG). They have proposed an algorithm for architectural model slicing through MDG traversal (AMSMT) to produce the static and dynamic architectural model slices. The algorithm traversed the edges of model dependency graph (MDG) according to the slicing criterion. They developed a tool which computed dynamic slices from UML architectural models.

Noda et al. [104] have extended their own work on dynamic slicing of the sequence diagram. They generated behavior model (B-Model), which defined various dependencies and calculated slices. They supported their work by incorporating exceptions and multithreading programs. They implemented a tool as Eclipse plug-in to demonstrate their proposed method.

3.3 Black-box Testing of Service-Oriented Software

In this section, we briefly present the reported work on black-box testing of web services, testing tools available for web services and model-based testing approaches, frameworks, and tools.

There exists a slight difference between web services and web applications. The web services are for machines or programs to machines or program communication over a
network whereas web application is for users. Web applications usually present data in HTML which looks nice to the user and web services usually present data in XML which easy to parse by other applications. A web application may contain both a graphical user interface for human users, as well as a set of web services for computer “users” (clients). For example, a payment service like Paypal has both a graphical user interface for human users, as well as a set of web services through which we can have own backend systems accessing the Paypal services.

The service-oriented technology hinders the application of practical traditional testing techniques, such as white-box approaches [49]. Bartolini et al. [49] introduced service-oriented coverage testing (SOCT), which probed the instrumented service code, and the service (tcov) to retrieve coverage related information through WSDL interface. The survey by Canfora et al. [84] gave good solutions to the challenges analyzed from the viewpoints of different stakeholders including different levels of testing i.e unit, integration, and regression testing. Motlagh [67] summarized literature on unit, integration, and regression testing in table formats. Canfora et al. [83] gave overview of SOA testings and fundamental testing issues such as changes in inputs, dynamic binding, service level agreement (SLA) violation, change in the functionality or quality of service (QoS) along with solutions for mutation testing, integration testing and regression testing. Kalamegam et al. [187] inspected recent research related to SOA testing. Patil [231] suggested various recommendation to test web services, interoperability testing, load testing, use of JMeter [6] to test web services, and some of the major web service testing tools.

The WSDL formalized descriptions of service operations and of their input and output parameters can be in fact taken as a reference for black-box testing at the service interface [33]. Many of the proposed approaches for WSDL-based data generation are based on the XML schema (XSD) data type information [15, 34, 40, 120, 210, 212, 250, 266]. In these approaches, the datatype with various constraints allows generation of test data. The complex types are decomposed into simple types to generate test data. Tsai et al. [237] have proposed an extension to WSDL description to find out various dependencies such as input-output dependency, invocation sequences, hierarchical functional descriptions, and concurrent sequence specification. The test data generation using WSDL is limited to the XSD constraints because it lacks run-time information of the services. As a result, many researchers have focused for other alternative specifications. The use of semantic model OWL-S for test data generation is proposed in [81, 246, 253, 257].

The existence of known errors can be detected by test cases, but the faults in system behaviour can not be determined through it. Fault-based test data generation aims at proving the absence of prescribed faults in web services. The fault-based test data
generation technique generates erroneous test data intentionally [120, 121].

Zhu [88] analyzed key challenges to test web services and proposed a service-oriented testing framework to test web services, where T-services are the testing services working on behalf of the customer and test broker services which search and invoke testers capable of performing testing on demand of the customers. Schieferdecker et al. [97] considered the applicability of testing and test control notation TTCN-3 for functional and load testing of web services. Mei [130] have proposed a context-aware framework for testing service orchestration and choreography in the presence of a context (runtime environment). In the work [206] various examples for white-box, black-box and gray box testing of services have been illustrated.

### 3.3.1 Testing Tools For Web Services

A few testing tools are available in market to test service-oriented software (SOS) based on web services. The most commonly used tools are:

- SmartBear SoapUI
- Crosschecknet SOAPSonar
- HP QuickTest Professional
- Parasoft SOAtest
- Apache JMeter
- Oracle Testing Accelerators for Web Services
- PushToTest TestMaker, and
- WebInject

All the above mentioned tools somehow support testing of web services from functional and nonfunctional aspects. Some of them offer an option for validation of WSDL document for conformance to the WS-I [244] profile. The organization for the advancement of structured information standards (OASIS) [176] is an industry consortium, which defined a set of rules showing how standards should be used and have designed various profiles related with web service interoperability. Also, it provides testing tools for use by the tester community to help in developing, deploying and testing interoperable web services.
- **SmartBear SoapUI**
  SoapUI [207] is an open-source testing tool for service-oriented architecture (SOA) and web service testing. It is developed by SmartBear software company and is provided freely under the GNU lesser general public license (LGPL). It provides complete test coverage and supports all the standard protocols and technologies. SoapUI facilitates quick creation of advanced performance tests and execution of automated functional tests of any test scenarios. The set of features offered by soapUI helps in performance evaluation of web services. It requires WSDL files to generate tests, messages, validations and mocking services. Unfortunately, soapUI supports only 1.1 version of WSDL, it can’t provide support for WSDL 2.0. SoapUI conforms WS-I profile [244] for 1.1 version of WSDL. The new version 1.7 includes plug-ins for the NetBeans, IntelliJ, and Eclipse IDEs. SoapUI can easily create functional, regression, compliance, and load tests. SoapUI provides code-free test environment. Analysis of the test results provides a mean to improve the quality of services and applications.
  Recently released version SoapUI NG Pro allows the option for flow to test variables, properties requests, and context. Test data for data-driven tests can be provided from external databases resources. SoapUI’s MockServices, a feature, which allows mimicking web services before they are developed. It removes the cost of building web service. MockServices can simulate any desired behavior of web service, no matter how complex, the service responses can be completely configured.

- **Crosschecknet SOAPSOnar**
  SOAPSOnar [208] is a specialized tool for testing and analyzing web services. It can perform regression tests, performance tests, generate reports and many other tests. The test case is developed via easy to use graphical interfaces. It supports both WSDL 1.1 and 2.0 documents. The professional edition of SOAPSOnar has test flow management options and can create a chain of WSDL requests-response or data-driven test case for the exchange of SOAP messages.
  Almost all the versions of SOAPSOnar uses test request and response analysis data from e.g. database tables. The application can automatically change values of variables in SOAP message headers, body, tasks and can change global variables and automation variables. SOAPSOnar has vulnerability mode, which associates each test request with a set of attack. Analyzing SOAP responses helps to find any existing vulnerabilities and leakage of information. It can parse the WSDL documents and automatically generate a list of the operations described by it. It can also be used to send SOAP request messages to the target web
service and capture the responses.

- **HP QuickTest Professional**

HP QuickTest Professional tool [89] is an automated testing suite. It uses both WSDL 1.1 and WSDL 2.0. It can perform automated testing on a variety of software and in varied environments. It allows regression and functional testing. The main functionality of HP QuickTest Professional is its ability to perform validation of WSDL using the WS-I tool [244]. It is quite complex but a more powerful tool compared to the other competitors.

- **Parasoft SOAtest**

SOAtest [181] is used for testing service-oriented architecture (SOA) based applications. It automates application testing, message/protocol testing, cloud testing, and security testing. Its graphical user interface is based on Eclipse. It allows creating tests, defining the behavior of the tests and configuring specific tests.

SOAtest’s main functionality is to test web services. It automatically generates tests from XML, XML schema definition (XSD), web service description language (WSDL), universal description discovery and integration (UDDI), WADL, WSIL, business process execution language (BPEL) and other. SOAtest can validate WSDL documents and can emulate the client or the server. Complex testing scenarios can be executed in a sequence. Failed test case are distinctively highlighted in the generated results. Parameters for the requests can be entered by the user or can be automated to be read from a file.

Even, SOAtest can trace and visualize how SOAP messages and events flow through enterprise service bus (ESB), message brokers, applications, and databases while tests are executed. Such tracing allows interpreting problems directly from the testing environment. SOAtest provides regression tests and functional tests. They will cause an alert when changes affect application’s behaviour. Functional tests like load testing can be performed for performance monitoring. Security penetration tests are performed at the message level. SOAtest is promising software to perform web services testing.

- **Apache JMeter**

JMeter [6] is an open-source testing tool for web services. It is developed by apache software foundation (ASF). It is distributed under Apache License. Initially, it was designed to test web applications but has been extended to test web
services. The core functionality of JMeter is to load test client/server applications/services. It can also be used for performance evaluation of applications. Additionally, JMeter helps in regression testing by facilitating creation of automated test scripts with predefined assertions. By this, we can verify the application’s results.

JMeter incorporated support for multithreading that allows concurrent functions execution by a number of threads. JMeter offers easy to use user interface. Configuration and setting up a testing plan is somewhat complex and requires little efforts. JMeter can generate a number of statistical reports and graphical results.

- **Oracle Testing Accelerators for Web Services**

Oracle application testing suite, includes testing accelerators for web Services. It allows testing quality and performance of service-oriented architectures based applications directly from web service interface level. It automates functional and regression testing of services and uses an OpenScript platform to allow users to generate scripts. These scripts can combine multiple simple object access protocol (SOAP) requests in a single test script. Web services test scripts can be created by selecting which methods to call from this store of parsed web service method requests or by specifying the requests manually. It can also specify data inputs for those web service requests using data banks. It can reuse response values from one request, parse them and then use it as input for subsequent requests of web services using script variables. However, it does not make any coverage analysis and suggests or create any test cases. It reports whether the test succeeds or passed.

- **PushToTest TestMaker**

TestMaker [182] is a web service testing application from PushToTest software company. TestMaker’s tests are embodied in scripts called test agents. It provides an agent wizard that will read a WSDL definition and automatically creates a skeletal test agent. TestMaker can also be used to test Web applications. It includes a network monitoring tool that can watch HTTP traffic between the browser and a target web application and generate test cases from this interaction. TestMaker test agents are written in Jython (python is written in Java). Jython can access all the Java libraries provided with TestMaker. It supports communication protocols: HTTP, HTTPS, SOAP, POP3, JDBC, and much more. Therefore, the tester can create test cases that approach any client application the web service is likely to be called by.
3.3.2 Model-based Testing of Service-Oriented Software

Model-based testing is a special type of black-box technique that offers many advantages over traditional testing techniques. First, the construction of design models can begin early in the development cycle and it allows improvement in the design too. Therefore maintenance cost can be minimized. Even the test cases are available early in the development cycle so it makes the test planning more effective. Since it finds faults at the initial stage of software development lifecycle, it reduces the testing time. Second, the modeling exposes ambiguities, inconsistencies, incompleteness and errors in the specification or design of software. Further, these model can be reused in future testing, even when the specifications changes rapidly. Moreover, these models are easier to update than a complete testing suite. The test data obtained from this technique is independent of the implementation architecture. Most importantly, these models embody information that can be used with graph theory to generate many different intermediate representations, test cases, test sequences, and test scenarios automatically.

Model-based testing requires that the models must be complete, consistent, unambiguous, and correct. Otherwise, testing will fail as the model is the core part. There are various challenges involved in model-based testing of SOA-based software. Some of these challenges are listed below:

- The web service description language (WSDL) is insufficient for testing as it contains only service description information, and the tester requires the complete business process model.
- Generally, we prefer UML for modeling static and dynamic behaviour of object-oriented software. Since SOA is heterogeneous and distributed and there does not exist any support for modeling such properties in UML, it complicates the testing process.
- There is no consensus about best modeling language for SOA. The object-oriented software is based on classes, objects, design patterns while service-oriented software has services, interface descriptions, business workflows etc.
In this section, we briefly present the reported work on model-based testing approaches, frameworks, and tools. Most of the work reported in the literature are focused on the development of testing techniques or frameworks based on models.

Once the analysis and requirement phase is completed the modeling phase starts. The most commonly used models are: finite state machines (FSMs), unified modeling language (UML), formal grammars, markov chains, petri nets, decision tables, decision trees, data flow diagrams, business process execution language (BPEL) and statecharts. Application of some of the models for testing web services can be found in the literature [16, 41, 42, 106, 122, 247]. Model-based testing techniques using petri nets have also been explored extensively in literature [48, 249, 259]. Cao et al. [226] have used a formal model, named timed extended finite state machine (TEFSM), defined by Lallali et al. [160, 161] to model web service composition. They developed web service online testing framework (WSOTF) to test a web service based on WSDL specification or a web service orchestration.

The business process execution language for web service (BPEL4WS), now BPEL, is an XML-based web service composition, orchestration, coordination and executable language for business processes in an SOA. A business process execution language for web service (BPEL4WS or BPEL) process is mainly composed of activities, which can be separated into basic activities, such as invoke, receive, reply, assign, throw, exist and structure activities, such as sequence, pick, if, while, repeatuntil, flow etc. The flow (\texttt{<flow>}) activity is used to define a set of activities that can be invoked in parallel. The web service composition promotes reusability by creating new web service capabilities through combining existing web service capabilities. The coordination in other hand refers to WS-coordination protocol or WS-business protocol which is used to denote the set of valid sequence of invocations that occur among the service participants during execution of the certain business process. Nowadays, business process execution language (BPEL) has become the de-facto standard for the compositions of web services.

Fu et al. [254] have presented a web service analysis tool (WSAT) for analyzing and verifying composite web service designs. The WSAT consists of a translator which converts business process execution language (BPEL4WS or BPEL) to an intermediate representation (GSFA), and spin model checker [85] to check LTL properties. Antonova et al. [62] have presented a tool named testing of service implementations (TESSI), which supported test case generation, execution and management. The core functionality of the TESSI tool includes generation of simple object access protocol (SOAP) request templates, definition of assertions at hypertext transfer protocol (HTTP), simple object access protocol (SOAP) and business process execution language (BPEL) variable levels, support of data driven testing, test execution and man-
Looker et al. [174] discussed software implemented fault injection (SWIFI) based tools. The SWIFI tool allowed specific systems running on target hardware to be effectively targeted without injecting faults into other parts of the system. Various tools for test input generation using dynamic symbolic execution have been researched extensively such as Pex [173], DART [188], Cute and Jcute [107]. DART [188] was the first tool for dynamic symbolic execution of C programs. Cute [107] is an extension of DART for handling concurrent programs, where it’s Java version is named Jcute. The recent tool, Pex is a white box test generator for .NET programs.

Unified modeling language (UML) is a standardized general purpose modeling language used in the field of engineering. UML diagrams represent two different views of a system model, static and dynamic. Static or structural view emphasizes the static structure of the system using objects, attributes, operations and relationships. The static view includes class diagrams and composite structure diagrams. Dynamic or behavioural view emphasizes the dynamic behaviour of the system by showing collaborations among multiple objects and changes to the internal states of objects. The dynamic view includes sequence diagrams, activity diagrams, and state machine diagrams. Many testing techniques have been reported towards the generation of test cases. The scenario-based techniques are based on coverage criteria. The model-based techniques identify respective test cases for the software using models such as UML diagrams. The path-based testing is based on static as well as dynamic control flow of the software. Static path testing is carried out by symbolic execution whereas dynamic path testing is carried out by executing the program. Finally, the goal-based techniques identify test cases covering a selected goal such as a statement or branch, irrespective of the path taken.

Samuel et al. [191] generated test cases from UML communication diagram. They generated communication tree from communication diagram (CD), performed post-order traversal on conditional predicates and apply function minimization technique to generate test data. They implemented a tool named UML behavioral test case generator (UTG) to demonstrate the generation of test cases.

Soldal et al. [142] defined a test generation scheme based on a conformance testing and a formal operational semantics for sequence diagrams, which takes input as sequence diagrams that may contain the operators \textit{assert} and \textit{neg} and that produces tests in the form of sequence diagrams. The algorithm is based on a formal operational semantics for sequence diagrams and is an adaptation of J. Tretmans et al. [98]. The operational semantics and the test generation algorithm are implemented in term rewriting language Maude defined by M. Clavel [135].

Prasanna et al. [136] have presented a model based approach for automated generation of test cases in object-oriented systems. They consider UML object diagram and
genetic algorithm’s tree crossover operator to generate new generation tree. The new generation of trees is converted into binary trees. Then, depth first search traversal is performed on binary trees to generate test cases.

Sharma et al. [144] have presented use case diagram graph (UDG) and sequence diagram graph (SDG) for generating test cases from use case and sequence diagrams respectively. They integrated UGD and SDG into system testing graph (STG). The STG is traversed to get test cases. They use a PIN authentication scenario of an ATM system. Also, they implemented a tool with the help of MagicDraw and Rational Rose.

Kundu et al. [52] have presented an approach for generating test cases from UML 2.0 activity diagrams with use case scope. They defined an activity path coverage criterion. Additionally, the generated test cases are capable of detecting various types of faults which may occur.

Swain et al. [200] illustrated a method to derive test cases from analysis artifacts such as use cases and their corresponding sequence diagrams. They generated test cases from use case dependency graph (UDG) and concurrent control flow graph (CCFG) from use case diagram and corresponding sequence diagram respectively. They generated test case using full predicate coverage criteria. They used library information system (LIS) to demonstrate their work.

Nayak et al. [3] have proposed an approach of synthesizing test data from information available from class diagrams, sequence diagrams and object constraint language (OCL) constraints. They generated a structured composite graph (SCG), incorporating system-wide information to generate test cases.

Shanthi et al. [1] focused on test case generation from UML sequence diagram using a genetic algorithm. They extracted information from sequence diagram and creates a sequence dependency table (SDT). With the help of SDT test path are generated. And then genetic algorithm are used to prioritize test cases.

Kosindrdecha [166] have proposed a technique to derive and generate tests from state chart diagram. They carried out an extensive literature survey and classified various test generation technique as specification-based techniques, sketch diagram-based techniques, and source code-based techniques. Grigorjevs et al. [102] have presented a testing technique for model generation from UML sequence diagram to UML Testing profile. They discussed principles of model transformation to generate test cases from sequence diagram and shows practical approach to specific model. Patnaik et al. [51] put an effort to represent deadlock situations with the help of graph in a distributed environment. They proposed an algorithm to detect deadlock with the help of real time banking system and generated test cases for it.

Panthi et al. [228, 229] generated test cases from sequence diagram automatically.
They named their approach as automatically test sequences generation from sequence diagram (ATGSD). In ATGSD, they first converted the sequence diagram into sequence graph then, the graph is traversed to select the predicate functions. Next, they transformed predicate into the source code. Then, they constructed extended finite state machine (EFSM) from the code. Finally, they generate test data corresponding to the transformed predicate functions and stored the generated test data for future use. They demonstrated their work based on bank ATM system and tool based on ModelJUnit library.

Sumalatha et al. [227] have presented a technique to generate test cases from integrating UML activity and sequence diagram. They used breadth first search on activity sequence graph which is generated from merging activity and sequence diagram. Priya et al. [199] generated test path from UML sequence diagram of a medical consultation system. Swain et al. [192] used condition slicing and generate test cases from UML interaction diagrams. They generated test cases from message flow dependence graph from UML sequence diagram and then applies conditioned slicing on a predicate node of the graph.

Meena [53] generated test cases from UML sequence diagram and interaction overview diagram. They transformed the sequence diagram and interaction overview diagram to an intermediate form called UML interaction graph (UIG) using XMI code. Then, UIG is traversed to find the all valid path and generate test cases. Kaur et al. [137] developed a methodology to derive test cases using conditional predicates.

The control flow graph (CFG) is an efficient intermediate representation or form of a program or software which is used to analyze various properties of a program and which in turn would be useful for software testing, software maintenance, and software measure or metrics. Also, CFG makes easier the understandability and comprehensibility of the source code which is more important for complex software like service-oriented software (SOS). A large number of CFG construction approaches have been proposed for many programming languages like procedural, object-oriented programming and aspect-oriented etc.

As stated earlier, model-based testing is considered as black-box testing but business process execution language (BPEL) testing, is performed in a white-box manner [185, 267]. Using business process execution language (BPEL), tester has access to the internal logic of the whole business process. As a result, tester can generate test cases according to the required coverage criteria.

Jehan et al. [214] have proposed a test case generation approach that combines constraints which are extracted from the business process execution language (BPEL) process activities directly with pre-conditions and post-conditions for called services. First, they transform a business process execution language (BPEL) program to a con-
trol flow graph. Then, they generated test data using the lp solver [163]. At last, they executed the test cases using BPEL Unit tool [61].

Alemneh et al. [66] have proposed a static code analysis tool SAT4BSC to compute metrics and draw control flow graph (CFG) for BPEL source code. Ouyang et al. [43] have proposed a tool WofBPEL, which performed static analysis on business process execution language (BPEL) processes by translating business process execution language (BPEL) processes into petri nets. This tool can detect conflicting messages, unreachable paths or activities and can serve as a garbage collector.

Yuan et al. [258], generated business flow graph (BFG) by converting business process execution language for web service (BPEL) code to a control flow graph (CFG). It’s just a specialization of control flow graph (CFG) for BPEL. Then they generated test cases from business flow graph (BFG). Endo et al. [17] have proposed a graph parallel control flow graph (PCFG) that is composed of multiple CFGs representing service composition and interactions among these CFGs and have also presented a tool that supports the proposed technique. Hou et al. [213] have proposed an approach which converts BPEL processes to message sequence graph (MSG) and then they generated test cases from this MSG. Other approaches for test case generation from BPEL processes is based on control flow graphs (CFGs), graph search algorithms, and constraint solvers can be found in the literature [17, 123, 258].

Mao [44] have proposed a method for static slicing of business process execution language (BPEL) programs for understanding and debugging purpose. They created an extended control flow graph (ECFG) based on elements \(<\text{flow}\>) and \(<\text{pick}\>) . Then, data dependence relations between service participants are computed. Based on this analysis, a dependence graph named as \(BPDG\) is built by introducing the concept of synchronized edge. Subsequently, the static (backward / forward) slicing algorithm is illustrated with the help of BusinessTravelProcess [162].

Yee [45] have presented a test case generation tool employing a data dependency for web services business process execution language (WS-BPEL or BPEL). In this approach, data dependencies are defined using XPath expression for a BPEL process under test (PUT). These dependencies along with Type definitions in WSDL documents are then used to automatically generate input test data and corresponding verification conditions. Finally, test cases are executed using these data. Nonchot et al. [46] focused on test case generation from a business process modeling notation (BPMN) with BPEL diagram and XSD schema (XSD). Lallai et al. [160, 161] extended the finite state machine as the web service time extended finite state machine (WS-TEFSM) to generate test cases that aim to exercise the time constraints in web service compositions i.e business process execution language (BPEL). They developed the tool BPEL2IF for the transformation of business process execution language (BPEL) to an
intermediate form, and TESTGen-IF to generate the test cases. Ma et al. [47] have proposed stream x-machine based testing techniques to BPEL and the stream x-machine generated test cases.

Linzhang et al. [248] have presented a technique that generated test cases from UML activity diagram using gray box method, where the model represented the expected structure and behavior of the system under test (SUT), and which were used for creating reused design. Then, all the contextual information for test case generation, i.e. input or output sequence and parameters, the constraint conditions and expected object method sequence, are derived from each test scenario. Finally, the possible values of all the input/output parameters are generated by applying category-partition method. They developed a prototype tool UMLTGF to support the testing process. They overcome the limitations of white-box testing technique which was unable to find all the potential paths. Then, test cases are generated which satisfied the path coverage conditions constructed by black-box testing. Their approach was able to resolve the problems associated with both black-box and white-box testing techniques.

Kaur et al. [186] used UML statechart diagrams to automatically derive test paths using graph coverage techniques. Additionally, they developed a tool which supported TestOptimal. From statecharts, model coverage graphs were constructed with the help of TestOptimal and then test paths were generated. The generated test paths covered node coverage, edge coverage, edge-pair coverage and prime path coverage. Further, they used chinese postman and prefix-based algorithms for manual and automatic generation of test cases. Bhattacharjee et al. [82] have presented a test case generation technique based on UML activity diagram and tabu search algorithm to find out the test path with the highest priority. Gupta [124] generated test scenarios automatically from activity diagram using depth search (DFS) method. In this technique, they generated a directed graph called activity flow graph (AFG) and an intermediate table called activity dependency table (ADT) automatically for each activity diagram. Automated test paths were generated from AFG and the test cases were generated to cover all the test paths of the AFG. Ashu Mishra [18] generated test sequences using UML activity diagram. Then, they used an ant colony optimization algorithm to prioritize test sequences.

Paradkar et al. [19] have proposed a model-based test data generation technique for semantic web services. In their approach, test cases were generated using pre-defined fault models and semantic specification. Florencio [125] developed a tool, SOA test accelerator (SOATA), to automate test creation in service-oriented architectures. Kumar [20] have proposed a model to get the automatic test cases for SOA based application. He applied different assertions on each service operation for generating the test cases. Kumar et al. [21] have proposed an automatic test data generation approach for
web services through XML schema. Their approach ensured the traceability of each element present in the XML schema by using XML tree table structure and further generated test cases.

Zhang et al. [268] extended UML 2.0 activity diagram to describe the syntax and behaviors of business process execution language (BPEL). They have mapped, business process execution language (BPEL) structure activity such as sequence, pick, if, while, repeatuntil, flow to UML 2.0 activity diagram. Next, they generated XML instances from the XML schema (XSD) using XML based partition testing (XPT) approach [15]. The XPT divided the input domain into a subdomain such that, within each of them there was no change in the program behavior. They implemented a tool, named testing by automatically generated XML instances (TAXI).

Ionita et al. [7] modeled services for water management system using SoaML service interface diagram. They discussed the usefulness of SoaML for communication and monitor water pollution on Romanian rivers. Ali et al. [169] have proposed Ambient-SoaML profile for mobile devices which changes location frequently. Fazziki et al. [8] have presented a process to align business requirement to service oriented architecture using BPMN, SoaML, and component model. Gebhart et al. [145] evaluated design decision such as loose coupling, high autonomy, discoverable etc. when designing a new service. Bhuyan et al. [183] have proposed a layered architecture which had the capability to analyze the developed services by comparing the users requirements using self-analysis database.

3.4 Summary

In this chapter, we have briefly reviewed the available work on slicing of object-oriented software and testing of SOA-based software relevant to our research. We have discussed the work on static and dynamic slicing of object-oriented program, slicing of UML, and SoaML models. Since there were no research reports available on static and dynamic slicing of SOA-based software, we have briefly reviewed static and dynamic slicing of object-oriented programs, architecture, slicing UML, and slicing of SoaML models. We also have discussed the reported literature for black-box and model-based testing of SOA-based software. As dynamic slicing algorithms are used in interactive applications such as debugging and testing, to meet this, we attempt to develop dynamic slicing algorithms and testing techniques for SOA-based software in the subsequent chapters in alignment with the goals of this thesis.